Aplicación de bases de Gröbner para programación entera y álgebra

Tesis presentada a la
Universidad Autónoma Metropolitana, Azcapotzalco
como requerimiento parcial para obtener el grado de
Doctor en Optimización
por

M. en C. Rodrigo Alexander Castro Campos

Asesores:

Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

Dr. Feliú Davino Sagols Troncoso
Departamento de Matemáticas, Cinvestav IPN

Ciudad de México, México
28 de abril de 2017

Agradecimientos

Mi primer agradecimiento va dirigido a las personas que se han tomado el tiempo de leer las líneas escritas en este trabajo, ya sea para proporcionarme su valioso consejo o para aprender un poco más del tema en el que trabajé durante tres años. Mi siguiente agradecimiento va dirigido a aquellas personas que tal vez no lean este trabajo, pero que en algún momento desearon sinceramente que lograra terminarlo con éxito. También quisiera agradecer a las personas que me han brindado alguna facilidad, por pequeña que haya sido, durante cualquier etapa en la realización de esta tesis: el trabajo con el que estoy por obtener mi máximo grado académico ha resultado mejor con su ayuda de lo que hubiera sido sin ella. Mis últimos agradecimientos son lo más difíciles, porque no es lo más adecuado describirlos con palabras. Estos van dirigidos a todos a los que les debo mucho de lo que soy como persona y profesionalmente, además de a las personas que han estado ofreciéndome su soporte en todo momento, ya sea desde que el momento en que me conocieron o incluso desde el momento en el que nací. Definitivamente mi vida sería muy diferente sin todos ustedes y es más fácil observar el resultado de todo su apoyo que poder describirlo en algunos pocos renglones.

Resumen

El cómputo de bases de Gröbner es una técnica cada vez más popular en la resolución de problemas planteados como sistemas de ecuaciones no lineales. Los algoritmos algebraicos que se usan para el cómputo de bases de Gröbner con frecuencia logran encontrar estructuras algebraicas ocultas en los sistemas, lo cual permite obtener todas las soluciones factibles del problema en un tiempo sorprendemente corto. Si un problema de optimización tiene un número finito de soluciones factibles, éste también puede resolverse usando bases de Gröbner. Adicionalmente, las bases de Gröbner han sido particularmente útiles en el análisis de sistemas criptográficos, logrando incluso vulnerar por primera vez algunos sistemas que habían resistido técnicas de ataque más comunes.

En esta tesis estudiamos el cómputo de bases de Gröbner para el problema de resolver sistemas de ecuaciones booleanas cuadráticas, el cual es computacionalmente difícil. Aunque los algoritmos más rápidos para calcular bases de Gröbner han cambiado poco en la última década, las implementaciones más rápidas y las únicas de interés práctico han sido hasta ahora propietarias. El trabajo desarrollado en esta tesis incluye la descripción y la implementación de una aplicación de código abierto que logra resolver sistemas booleanos no triviales en tiempos comparables con las implementaciones propietarias.

Por otra parte, las bases de Gröbner son de gran interés teórico ya que deben satisfacer un conjunto de propiedades algebraicas importantes. Para sistemas de ecuaciones con poca estructura, el problema de encontrar una base de Gröbner suele ser más difícil que el problema de encontrar todas las soluciones factibles del sistema. En esta tesis describimos dos algoritmos que calculan bases de Gröbner partiendo del conjunto de soluciones (o de no soluciones) de un sistema booleano. Para sistemas con pocas variables, el cálculo de las soluciones por fuerza bruta seguido de la aplicación de los algoritmos desarrollados en este trabajo es más eficiente que el uso de los algoritmos algebraicos tradicionales.

Índice general

Índice de figuras

Índice de cuadros

Lista de algoritmos

Capítulo 1

Introducción

El cómputo de bases de Gröbner generaliza a la eliminación gaussiana para el caso de sistemas de ecuaciones no lineales de varias variables. Al igual que la eliminación gaussiana en el caso lineal, las bases de Gröbner permiten encontrar una representación única y reducida de un sistema no lineal, de manera que sea posible listar todas sus soluciones mediante el proceso de sustitución hacia atrás. Para sistemas con un número finito de soluciones y una función objetivo asociada, se puede guiar al proceso para listar primero la solución óptima. El uso de restricciones no lineales permite restringir un sistema a sólo admitir soluciones enteras, por lo que las bases de Gröbner también pueden resolver problemas de programación entera.

En esta tesis nos concentramos en el cálculo de bases de Gröbner booleanas como método de resolución para el caso especial de sistemas de ecuaciones con variables y aritmética en $\mathbb{Z}_2$. En la práctica, esta decisión no es particularmente problemática. En este capítulo explicamos cómo transformar eficientemente un problema de programación entera en un sistema no lineal, que es resoluble usando bases de Gröbner booleanas. Posteriormente damos la definición formal de bases de Gröbner booleanas y explicamos cómo calcularlas usando el algoritmo de Buchberger. También describimos los criterios de Buchberger, los cuales sirven para predecir cálculos redundantes que aparecen durante el algoritmo.

1.1. Preliminares de álgebra elemental

Un semigrupo es un conjunto cerrado bajo un operador binario asociativo $\cdot$. Un monoide es un semigrupo que posee un elemento identidad e tal que $m \cdot e = e \cdot m = e$ para todo elemento m del monoide. Si además existe para cada m un elemento inverso m^{-1} tal que $m \cdot m^{-1} = m^{-1} \cdot m = e$, el monoide se denomina grupo. Un grupo es abeliano si el operador $\cdot$ es conmutativo. Un anillo es un conjunto equipado con dos operadores $+, \cdot$ donde el operador

$+$ induce un grupo abeliano, cuyo elemento identidad se denota como 0, mientras que el operador $\cdot$ se distribuye sobre $+$ e induce un monoide, cuyo elemento identidad se denota como 1. Usualmente se denota como $-m$ al inverso de m bajo el operador $+$ y se puede definir un operador $-$ tal que $m_1 - m_2 = m_1 + (-m_2)$. A su vez, el predicado binario $m_1 \mid m_2$ es verdadero si y sólo si existe un m' tal que $m' \cdot m_1 = m_1 \cdot m' = m_2$, en cuyo caso se dice que m_1 divide a m_2 o bien, que m_1 es un divisor de m_2 y que m_2 es un múltiplo de m_1. Un común divisor de dos elementos m_1 y m_2 diferentes de cero es un m' que es divisor tanto de m_1 como de m_2. Un común múltiplo se define de manera similar. El máximo común divisor de m_1 y m_2 se denota por $mcd(m_1, m_2)$ y es un común divisor que es múltiplo de cualquier otro común divisor. El mínimo común múltiplo se denota por $mcm(m_1, m_2)$ y es un común múltiplo que es divisor de cualquier otro común múltiplo. Un ideal es un subconjunto de un anillo tal que el operador $+$ induce un grupo y éste contiene todos los $m_1 \cdot m_2$ donde m_1 pertenece al anillo y m_2 al ideal. Un campo es un anillo en el que el operador $\cdot$ induce un grupo abeliano sobre los elementos diferentes de cero. Usualmente se define un operador $\div$ tal que $\frac{m_1}{m_2} = m_1 \cdot m_2^{-1}$. Además, para cualquier pareja de elementos m_1 y m_2 diferentes de cero, tanto el máximo común divisor como el mínimo común múltiplo existen y son únicos.

Un polinomio es una suma de productos de potencias enteras no negativas de un conjunto de variables $\{x_1, x_2, \ldots, x_n\}$ multiplicados por coeficientes de un campo K. Se denota por $K[x_1, x_2, \ldots, x_n]$ al conjunto de todos los polinomios anteriores. Un monomio es un polinomio de un solo sumando. Sea m un monomio, denotaremos por $coef(m)$ al coeficiente de m y por $x_i(m)$ al exponente de x_i en m. El grado de un monomio m está dado por $deg(m) = \sum_{i=1}^{n} x_i(m)$. Si $deg(m) = 0$, entonces el producto de sus variables es vacío y $m = coef(m)$. El grado de un polinomio p se denota como $deg(p)$ y es el mayor grado de sus monomios. Un polinomio es homogéneo si todos sus monomios tienen el mismo grado. Un polinomio es lineal, cuadrático o cúbico si su grado está acotado por arriba por 1, 2 o 3, respectivamente. Una ecuación es polinomial lineal, cuadrática o cúbica si consiste en una constante igualada a un polinomio lineal, cuadrático o cúbico, respectivamente. Denotamos como $p(x)$ a la evaluación de p con una asignación x de valores para sus variables.

La suma de monomios $m_3 = m_1 + m_2$ está definida si $x_i(m_1) = x_i(m_2)$ para $1 \leq i \leq n$ y está dada por $coef(m_3) = coef(m_1) + coef(m_2)$ y $x_i(m_3) = x_i(m_1)$ para $1 \leq i \leq n$. La suma de polinomios $p_1 + p_2$ es un polinomio que incluye los monomios tanto de p_1 como de p_2. El producto de monomios $m_3 = m_1 \cdot m_2$ está dado por $coef(m_3) = coef(m_1) \cdot coef(m_2)$ y $x_i(m_3) = x_i(m_1) + x_i(m_2)$ para $1 \leq i \leq n$. El producto de un polinomio por un monomio es la suma de los productos de sus monomios por el factor. La división de monomios $m_3 = \frac{m_1}{m_2}$ está definida si todos los exponentes $x_i(m_3) = x_i(m_1) - x_i(m_2)$ son no negativos para $1 \leq i \leq n$,

siendo $coef(m_3) = \frac{coef(m_1)}{coef(m_2)}$. En este caso, se dice que m_2 es un divisor de m_1.

Un polinomio p con variables $x_1, x_2, \ldots, x_n$ está simplificado si para cualquier elección de exponentes $e_1, e_2, \ldots, e_n$ existe a lo mucho un monomio m en p tal que $x_i(m) = e_i$ para $1 \leq i \leq n$. Se denotará como $x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}(p)$ al único monomio de un polinomio simplificado p con tal producto de potencias de variables. Se asume que $e_i = 0$ si x_i no aparece bajo la notación anterior para $1 \leq i \leq n$. Si para alguna elección de exponentes no existe en p el monomio correspondiente, la expresión $x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}(p)$ puede definirse como $0 \cdot x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}$. Si $coef(x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}(p)) = 0$ para cualquier elección de exponentes, se dice que $p = 0$.

1.2. Preliminares de complejidad computacional

Informalmente, un algoritmo es una lista finita de instrucciones elementales que llevan a cabo una tarea finita. Un algoritmo toma una entrada y eventuamente produce una salida, las cuales pueden ser objetos arbitrarios o incluso ser vacías. Por otra parte, para que un algoritmo con entrada o salida no vacías pueda ser ejecutado sobre una máquina abstracta, éstas deben presentarse codificadas como cadenas finitas de símbolos, usualmente bits. La longitud de una entrada o salida es el número de símbolos de su representación. Por ejemplo, n bits son suficientes para codificar cualquier entero no negativo menor que 2^n.

El tiempo de ejecución $f_A(n)$ de un algoritmo A es el máximo número de instrucciones elementales que el algoritmo ejecuta para una entrada de longitud n. Dado que el valor exacto puede ser difícil de calcular, se acostumbra determinar el orden de una función de n que acote por arriba al tiempo de ejecución. La notación $\mathcal{O}$ es la más usual para este fin. Se dice que $f_A(n) = \mathcal{O}(g(n))$ si para toda $n \geq n_0$ se cumple que $f_A(n) \leq c \cdot g(n)$ para alguna elección de constantes $n_0 \in \mathbb{N}$, $c \in \mathbb{R}$. Se dice que A es un algoritmo polinomial si g también lo es. Lo análogo ocurre si g tiene otro orden tal como lineal, exponencial o factorial.

Un problema de decisión es uno cuya respuesta es *sí* o *no*. Por ejemplo, ¿el entero n tiene raíz cuadrada entera? Un problema pertenece a la clase $\mathcal{NP}$ si es un problema de decisión y además existe un algoritmo no determinístico que lo resuelve en tiempo de ejecución polinomial al tamaño de la entrada. Si un problema en $\mathcal{NP}$ tiene un algoritmo determinístico que lo resuelve en tiempo polinomial, el problema pertenece a la subclase $\mathcal{P}$. Se dice que un problema Q es $\mathcal{NP}$-Completo si pertenece a $\mathcal{NP}$ y toda entrada de cualquier problema Q' en $\mathcal{NP}$ puede ser reescrita como una entrada de Q en tiempo polinomial, de tal modo que existe una correspondencia entre la salida de los algoritmos que resuelven Q y Q'. Se dice que un problema Q (no necesariamente de decisión) es $\mathcal{NP}$-Duro si toda entrada de cualquier problema Q' en $\mathcal{NP}$ puede ser reescrita como una entrada de Q en tiempo polinomial, de tal

modo que existe una correspondencia entre la salida de los algoritmos que resuelven Q y Q'. Existen algoritmos exponenciales para todos los problemas $\mathcal{NP}$-Completos y se cree que no existen algoritmos polinomiales para ningún problema $\mathcal{NP}$-Completo [47].

1.3. Preliminares de álgebra lineal

Un sistema de polinomios lineales $p_1, p_2, \ldots, p_m$ distintos de cero se puede representar en forma matricial de la siguiente manera. Sea $\{x_1, x_2, \ldots, x_n\}$ el conjunto de variables del sistema. Se construye una matriz A de m filas y $n+1$ columnas donde las entradas de la matriz están dadas por $A_{i,j} = \mathit{coef}(x_j^1(p_i))$ y $A_{i,n+1} = \mathit{coef}(x_1^0 x_2^0 \ldots x_n^0(p_i))$. Sea la columna líder $lc(A_i)$ la menor j en $1 \leq j \leq n+1$ con $A_{i,j} \neq 0$. Una matriz A está en forma escalonada por filas si $lc(A_i) \neq lc(A_k)$ para $1 \leq i \neq k \leq m$. Una matriz A está en forma escalonada reducida por filas si $A_{i,lc(A_k)} = 0$ para $1 \leq i \neq k \leq m$.

El algoritmo de la eliminación gaussiana transforma una matriz A a su forma escalonada reducida de la siguiente forma. Sean i, k enteros en $1 \leq i \neq k \leq m$ tales que $A_{i,lc(A_k)} \neq 0$. La fila A_i es reemplazada por $A_i - \frac{A_{i,lc(A_k)}}{A_{k,lc(A_k)}} \cdot A_k$. El proceso se repite mientras exista alguna pareja que cumpla la condición o alternativamente, hasta que la matriz llegue a una forma escalonada si no se requiere llegar a la forma escalonada reducida. Al terminar, las filas sin coeficientes distintos de cero se eliminan y las restantes se permutan de modo que $i < j \Rightarrow lc(A_i) < lc(A_j)$. Todas las formas escalonadas de una matriz tienen el mismo conjunto de columnas líderes $\{lc(A_i) : 1 \leq i \leq m\}$ y la forma escalonada reducida es única. La eliminación gaussiana puede realizarse en tiempo $\mathcal{O}(\text{máx}(m, n)^3)$ independientemente de si el proceso se detiene al llegar a una forma escalonada que no sea la forma escalonada reducida [37].

Un sistema de ecuaciones lineales es equivalente al sistema denotado por su matriz escalonada reducida. Sea A una matriz escalonada reducida de m filas y $n+1$ columnas. Si existe alguna i en $1 \leq i \leq m$ tal que $lc(A_i) = n+1$ entonces el sistema no tiene solución. En caso contrario, el proceso de sustitución hacia atrás permite encontrar todas sus soluciones de la siguiente manera. Calcule el conjunto de soluciones S_m de la ecuación $p_m = 0$ donde p_m es el polinomio representado por la fila A_m, dejando indefinidos los valores de las x_i tales que $\mathit{coef}(x_i(p_m)) = 0$ para $1 \leq i \leq n$. Para cada una de las soluciones $s \in S_m$, fije los valores de las x_i definidas en s y repita el proceso recursivamente sobre la submatriz $A_{1:m-1,1:n+1}$. El proceso termina cuando la submatriz queda vacía y la secuencia de valores fijados durante la rama actual de la recursión corresponde a una de las soluciones del sistema completo. Las variables que permanecieron indefinidas al terminar el proceso pueden tomar cualquier valor. En general, las decisiones siempre pueden tomarse en el orden $x_n, x_{n-1}, \ldots, x_1$ [37].

1.4. Preliminares de programación entera

El problema de programación lineal consiste en optimizar una función lineal que está sujeta a un conjunto de restricciones lineales. La optimización puede consistir en maximizar o minimizar el valor de la función objetivo según sea el caso. Un problema de programación lineal es entero cuando todas las variables están restringidas a tomar valores enteros.

Problema: Programación lineal entera **Entrada**: Números $m, n \in \mathbb{N}$, una matriz $A \in \mathbb{Q}^{m \times n}$, vectores $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$ **Salida**: Un vector $x \in \mathbb{Z}^n$ tal que $Ax \leq b$ y $c^\top x$ es óptimo

Si la función objetivo $c^\top x$ se reescribe como la restricción $z - c^\top x = 0$ donde z es una variable artificial, un problema de programación lineal entera puede replantearse como el problema de encontrar una solución factible con el menor o el mayor valor de z según sea el caso [3]. Resolver un programa lineal entero es un problema $\mathcal{NP}$-Duro, aún cuando las variables sean binarias y sólo se busque determinar factibilidad [35]. Por otra parte, es posible transformar un programa lineal entero en un programa que sólo utilice variables binarias cuando las variables del programa original están acotadas por una constante. Suponga que $0 \leq x_i \leq L$ para toda $1 \leq i \leq n$. El programa lineal binario puede construirse como sigue:

- Defina variables binarias de la forma $b_{(x_i,0)}, b_{(x_i,1)}, \ldots, b_{(x_i,\lfloor \log_2(L) \rfloor)}$ para toda $1 \leq i \leq n$. El valor del entero x_i estará expresado implícitamente como $\sum_{j=0}^{\lfloor \log_2(L) \rfloor} 2^j \cdot b_{(x_i,j)}$.

- Construya la matriz de restricciones del nuevo programa sustituyendo las x_i del programa original por la expresión arriba mencionada, para toda $1 \leq i \leq n$.

En el caso de que x_i pueda tomar valores negativos y $|x_i| \leq L$, se debe generar una variable binaria adicional que contribuya con un peso negativo dentro de la sumatoria. El valor de x_i puede expresarse como $\sum_{j=0}^{\lfloor \log_2(L) \rfloor} 2^j \cdot b_{(x_i,j)} - 2^{\lfloor \log_2(L) \rfloor + 1} \cdot b_{(x_i,\lfloor \log_2(L) \rfloor + 1)}$, lo cual tiene la propiedad de que existe una única manera de sumar un valor específico de x_i. La complejidad de la transformación es $\mathcal{O}(m \cdot n \cdot \log_2(L))$, que es polinomial con respecto al tamaño de la entrada. En la práctica, el valor de $\log_2(L)$ es del orden de las decenas.

La misma idea puede usarse si la función objetivo se reescribe como restricción con $|z| \leq L$. Si se tienen todas las soluciones factibles del problema, la búsqueda de la solución óptima puede guiarse por los valores de $b_{(z,\lfloor \log_2(L) \rfloor + 1)}, b_{(z,\lfloor \log_2(L) \rfloor)}, \ldots, b_{(z,0)}$ en ese orden. La magnitud de las contribuciones de las variables decrece exponencialmente, con $2^i \cdot b_{(z,i)} > \sum_{j=0}^{i-1} 2^j \cdot b_{(z,j)}$.

1.5. Preliminares de álgebra booleana

El álgebra booleana estudia las operaciones que pueden llevarse a cabo cuando sólo se permiten variables y constantes binarias. Los enteros 0 y 1 tienen una correspondencia con los valores de verdad *falso* y *verdadero*. Al conjunto de estos valores también se le conoce como el dominio booleano, el cual es denotado por $\mathbb{B}$. Los operadores más comunes son la conjunción $\wedge$, la disyunción $\vee$, la disyunción exclusiva $\oplus$ y la negación $\neg$. Ver cuadro 1.1.

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

x	$\neg x$
0	1
1	0

Cuadro 1.1: Tablas de verdad de operadores booleanos

La precedencia de la conjunción es mayor que la de la disyunción. La disyunción exclusiva es equivalente a $x \wedge \neg y \vee \neg x \wedge y$. Estos operadores además tienen un equivalente aritmético: el valor de $x \wedge y$ coincide con el valor del producto xy mientras que $\neg x$ es equivalente a $1 - x$. Además, $x \oplus y \equiv (x + y) \pmod 2$. Por completitud notaremos que $1 \equiv -1 \pmod 2$.

Cuando una fórmula booleana se vuelve verdadera bajo una asignación de valores de verdad para las variables, decimos que dicha asignación es una solución de la fórmula. Dada una fórmula booleana, el problema de encontrar alguna de sus soluciones es $\mathcal{NP}$-Duro [13].

Problema: Satisfacibilidad booleana
Entrada: Un número $n \in \mathbb{N}$, una fórmula booleana b con n variables.
Salida: Un vector $x \in \mathbb{B}^n$ tal que $b(x)$ es verdadera.

Dos fórmulas booleanas pueden ser sintácticamente diferentes y aún así ser equivalentes al tener el mismo conjunto de soluciones. Por esta razón, generalmente no se estudian fórmulas booleanas en formas sintácticas arbitrarias sino que primero se reescriben en alguna forma sintáctica convencional. La terminología para describir estas formas sintácticas es la siguiente.

Se le denomina literal a una variable o constante booleana, opcionalmente negada (en cuyo caso la literal es negativa). Una cláusula es un conjunto de literales conectadas por disyunciones y un término es un conjunto de literales conectadas por conjunciones. Una claúsula o término son positivos si no tienen literales negativas. Una fórmula está en forma normal conjuntiva si está escrita como un conjunto de cláusulas conectadas por conjunciones. Si ninguna cláusula tiene más de k literales, la forma se denomina k-normal conjuntiva.

Problema: Satisfacibilidad booleana en forma 3-normal conjuntiva
Entrada: Un número $n \in \mathbb{N}$, una fórmula booleana b con n variables escrita en forma 3-normal conjuntiva.
Salida: Un vector $x \in \mathbb{B}^n$ tal que $b(x)$ es verdadera.

El problema anterior también es $\mathcal{NP}$-Duro [35]. Es posible transformar una fórmula en forma normal conjuntiva a una en forma 3-normal conjuntiva en tiempo polinomial mediante la introducción de variables auxiliares. Por ejemplo, la cláusula $l_1 \vee l_2 \vee l_3 \vee l_4$ se puede reescribir como $(x_1 \vee l_3 \vee l_4) \wedge (\neg x_1 \vee l_1 \vee l_2) \wedge (x_1 \vee \neg l_1) \wedge (l_1 \vee \neg l_2)$ que está en forma 3-normal conjuntiva. De esta manera, una cláusula con k literales se puede reescribir en tiempo $\mathcal{O}(k)$. A su vez, es posible transformar el conjunto de m restricciones de un programa lineal binario con n variables en una fórmula en forma normal conjuntiva. Suponga que $|C| \leq L$ para todo coeficiente C que aparece en el programa. Un algoritmo que produce una fórmula con $\mathcal{O}(m \cdot n \cdot \log_2(L))$ variables en tiempo óptimo está descrito en [46].

Algunas veces es conveniente usar disyunciones exclusivas en lugar de disyunciones ordinarias, pues la estructura $(\mathbb{B}, \wedge, \oplus)$ es equivalente a $(\mathbb{Z}_2, *, +)$. Para lograr lo anterior, la fórmula generalmente se reescribe a la forma normal algebraica. Una fórmula está en forma normal algebraica si está escrita como un conjunto de términos positivos conectados por disyunciones exclusivas. La reescritura en tiempo polinomial de una fórmula en forma 3-normal conjuntiva con variables $x_1, x_2, \ldots, x_n$ a la forma normal algebraica casi siempre requiere la introducción de nuevas variables $w_1, w_2, \ldots, w_n$ y puede realizarse de la siguiente manera:

- Agregar a la conjunción n fórmulas de la forma $x_i \oplus w_i$ para $1 \leq i \leq n$.
- Reemplazar cada literal negativa de la forma $\neg x_i$ por w_i.
- Rescribir las cláusulas $l_1 \vee l_2 \vee l_3$ como $l_1 \wedge l_2 \wedge l_3 \oplus l_1 \wedge l_2 \oplus l_1 \wedge l_3 \oplus l_2 \wedge l_3 \oplus l_1 \oplus l_2 \oplus l_3$.

Después de la transformación, la fórmula se convierte en una conjunción de fórmulas en forma normal algebraica. Esto también puede verse como un sistema de ecuaciones binarias (con coeficientes y variables en $\mathbb{Z}_2$) no lineales e igualadas a uno. La conversión a un sistema igualado a cero es trivial. El problema de encontrar alguna de las soluciones de un sistema de ecuaciones binarias no lineales es $\mathcal{NP}$-Duro [1].

Problema: Ecuaciones binarias
Entrada: Números $m, n \in \mathbb{N}$, ecuaciones $(f_1, f_2, \ldots, f_m) \in \mathbb{Z}_2[x_1, x_2, \ldots, x_n]^m$ con variables $x_1, x_2, \ldots, x_n \in \mathbb{Z}_2$.
Salida: Un vector $x \in \mathbb{Z}_2^n$ tal que $f_1(x) = f_2(x) = \cdots = f_m(x) = 0$.

1.6. Preliminares de bases de Gröbner booleanas

Las técnicas algebraicas usadas en la resolución de sistemas de ecuaciones no buscan explícitamente encontrar una solución, sino que buscan encontrar un sistema de ecuaciones equivalente para el que sea trivial obtener todas sus soluciones. El cómputo de bases de Gröbner toma como entrada un conjunto de polinomios no necesariamente lineales (correspondientes a las ecuaciones de un sistema igualado a cero) y produce el conjunto de polinomios equivalente.

Aunque en la práctica las bases de Gröbner casi siempre se calculan para conjuntos de polinomios con coeficientes y variables racionales o complejas, los algoritmos usados también funcionan cuando los coeficientes y variables están restringidos a $\mathbb{Z}_k$ con k primo. Cuando $k = 2$, se dice que los polinomios y las bases correspondientes son booleanas. A continuación se presentan los conceptos necesarios para definir formalmente las bases de Gröbner booleanas.

Un polinomio booleano es un elemento de $\mathbb{Z}_2[x_1, x_2, \ldots, x_n]$ restringido por las ecuaciones de campo $x_i^2 - x_i = 0$ para $1 \leq i \leq n$. Por simplicidad notacional, se dice que un polinomio booleano es un elemento del anillo $\mathbb{B}(x_1, x_2, \ldots, x_n)$. Dos monomios m_1, m_2 son coprimos si $mcd(m_1, m_2) = 1$. Un orden monomial booleano es un orden total entre monomios booleanos tal que 1 es el elemento mínimo del orden y $m_1 < m_2 \Rightarrow m_1 m_3 < m_2 m_3$ cuando m_2 y m_3 son coprimos, para cualesquiera m_1, m_2, m_3. Dado un orden monomial fijo, el monomio líder de un polinomio p se denota por $lm(p)$ y es el mayor monomio de p en el orden. Un conjunto $G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$ es una base generadora del ideal $\mathcal{I} = \{g_1 \cdot q_1 + \cdots + g_k \cdot q_k\}$ donde $g_i \in G$ y $q_i \in \mathbb{B}(x_1, x_2, \ldots, x_n)$. Al ideal generado por G se le denota como $\mathcal{I}(G)$. Si para toda $p \in \mathcal{I}$ existe $g \in G$ tal que $lm(g) \mid lm(p)$, se dice que G es una base de Gröbner de $\mathcal{I}$.

Problema: Bases de Gröbner booleanas **Entrada**: Un número $n \in \mathbb{N}$, un conjunto $G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$, orden monomial $<$ **Salida**: Una base de Gröbner booleana de G bajo el orden $<$.

En 1965, Buchberger introdujo una caracterización de las bases de Gröbner y el primer algoritmo para calcularlas [9]. Sean $p, q \in \mathbb{B}(x_1, x_2, \ldots, x_n)$, la reducción de p por q está definida como $red(p, q) = p - \frac{lm(p)}{lm(q)} q$ si se cumple que $lm(q) \mid lm(p)$. Sea $Q \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$, la reducción de p por Q consiste en reducir p sobre cualquier $q \in Q$ para el cual la reducción esté definida, reasignar p como el resultado de dicha reducción y repetir el proceso mientras sea posible. El s-polinomio de $p, q \in \mathbb{B}(x_1, x_2, \ldots, x_n)$ está definido como $sp(p, q) = \frac{\lambda_{p,q}}{lm(p)} p - \frac{\lambda_{p,q}}{lm(q)} q$ donde $\lambda_{p,q} = mcm(lm(p), lm(q))$. Sea $G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$ una base generadora de $\mathcal{I}$, G es una base de Gröbner de $\mathcal{I}$ si para toda $g_i, g_j \in G$ se cumple que $red(sp(g_i, g_j), G) = 0$.

La idea principal del algoritmo de Buchberger es agrandar el conjunto de entrada G con los residuos diferentes a cero que resulten de la reducción de s-polinomios de G. Estos residuos

pertenecen al ideal pero no tienen un reductor en la G actual. En el caso del cálculo de bases de Gröbner booleanas, las ecuaciones de campo se consideran parte del G inicial. Buchberger también ideó dos criterios que predicen si un s-polinomio reducirá a cero, pues las reducciones a cero no contribuyen a agrandar G y pueden descartarse [8].

Teorema 1.1 (Criterio de coprimalidad)**.** *Sean $p, q \in G$ y $G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$. El s-polinomio $sp(p, q)$ reduce a cero si $lm(p)$ y $lm(q)$ son coprimos.*

Teorema 1.2 (Criterio de la cadena)**.** *Sean $p, q \in G$ y $G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$. El s-polinomio $sp(p, q)$ reduce a cero si existe $g \in G$ tal que $red(sp(p, g), G) = red(sp(q, g), G) = 0$ y $lm(g) \mid mcm(lm(p), lm(q))$.*

Algoritmo 1.1 Algoritmo de Buchberger

subrutina BUCHBERGER($G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$)
  $Q \leftarrow \{(p_i, p_j) : p_i, p_j \in G \wedge i < j\}$
  mientras $Q \neq \emptyset$
    $q \leftarrow selecciona(Q), Q \leftarrow Q \setminus q$
    si $\neg criterio_1(q_1, q_2) \wedge \neg criterio_2(G, q_1, q_2)$ **entonces**
      $r \leftarrow red(sp(q_1, q_2), G)$
      si $r \neq 0$ **entonces**
        $Q \leftarrow Q \cup \{(r, g) : g \in G\}$
        $G \leftarrow G \cup r$
  regresa G

La subrutina *selecciona* puede escoger cualquier elemento de Q, el cual actúa como una cola de parejas de polinomios para las que su s-polinomio falta por procesarse. La base de Gröbner de un conjunto de polinomios generalmente no es única. Además de que las bases suelen diferir bajo órdenes monomiales distintos, pueden existir varias bases para un mismo orden. Sin embargo, la que se conoce como base reducida es única para el orden dado y suele ser la más compacta. Sean $p, q \in \mathbb{B}(x_1, x_2, \ldots, x_n)$, se dice que q reduce la cola de p si para algún $m \in p \neq lm(p)$ se tiene que $lm(q) \mid m$. La reducción de cola está dada por $p - \frac{m}{lm(q)}q$. Una base de Gröbner G está reducida si ningún $g_i \in G$ es cola-reducible por algún $g_j \in G \setminus g_i$.

Las ecuaciones de campo $x_i^2 - x_i = 0$ implican que $x_i \in \{0, 1\}$, por lo que $x_i^k = x_i$ para $k \geq 1$. Al simplificar un polinomio de esta manera, ninguna variable tendrá un exponente mayor que uno y el valor de n se vuelve una cota superior de su grado. Durante el algoritmo de Buchberger, es válido aplicar esta simplificación a los elementos de G que no sean ecuaciones de campo, a los s-polinomios y a los polinomios intermedios y finales de una reducción. Los s-polinomios entre dos ecuaciones de campo pueden descartarse ya que reducen a cero por

el criterio de coprimalidad. Por otra parte, el s-polinomio simplificado entre un polinomio y una ecuación de campo puede obtenerse usando el siguiente resultado [34].

Lema 1.1. *Sea $p \in \mathbb{B}(x_1, x_2, \ldots, x_n)$. El s-polinomio $sp(p, x_i^2 - x_i)$ se simplifica a $x_i \cdot p$ bajo las ecuaciones de campo.*

La complejidad de calcular una base de Gröbner booleana es $\mathcal{O}(n \cdot 2^{3n})$ para cualquier elección de orden monomial [21]. Por otra parte, algunas propiedades de la base de Gröbner dependen del orden monomial elegido. Dos de los órdenes más utilizados son el orden lexicográfico y el orden graduado lexicográfico reverso. El orden lexicográfico es un orden total tal que $m_1 < m_2 \Rightarrow x_i(m_1) < x_i(m_2)$ para la menor i en $1 \leq i \leq n$ tal que $x_i(m_1) \neq x_i(m_2)$. Este orden se usa cuando se desea que la base de Gröbner resultante tenga una estructura que permita usar el proceso de sustitución hacia atrás [15]. El orden graduado lexicográfico reverso es un orden total tal que $m_1 < m_2 \Rightarrow deg(m_1) < deg(m_2) \vee (deg(m_1) = deg(m_2) \wedge x_i(m_1) > x_i(m_2))$ para la mayor i en $1 \leq i \leq n$ tal que $x_i(m_1) \neq x_i(m_2)$. En la práctica, el cálculo de una base bajo este orden suele terminar mucho más rápido que el cálculo usando el orden lexicográfico [2]. Si se desea obtener una base de Gröbner bajo el orden lexicográfico, generalmente es más eficiente usar el orden graduado lexicográfico reverso y luego realizar una conversión de orden, la cual toma tiempo $\mathcal{O}(n \cdot d^3)$ donde d es el número de soluciones del sistema [25, 38].

Capítulo 2

Estado de la técnica

En este capítulo discutimos la implementación de los criterios de Buchberger para detectar s-polinomios redundantes y estudiamos las implementaciones disponibles del algoritmo de Buchberger que se especializan en el cálculo de bases de Gröbner booleanas. Posteriormente introducimos el algoritmo F4, el cual es una modificación del algoritmo de Buchberger que usa técnicas de álgebra lineal para acelerar el cálculo. También presentamos el algoritmo F5, el cual introduce por primera vez la estrategia de firmas de polinomios para intentar predecir aún más s-polinomios redundantes. Al final de la sección describimos brevemente el sistema criptográfico HFE y la instancia conocida como el Reto HFE 1, el cual es de importancia histórica al haber sido resuelto por primera vez usando bases de Gröbner booleanas.

2.1. Implementaciones de los criterios de Buchberger

Para implementar el criterio de coprimalidad, basta con inspeccionar los monomios líderes de la pareja de polinomios dada. Por otra parte, la implementación del criterio de la cadena necesita determinar si otras parejas de polinomios relacionadas ya han sido consideradas (ya sea por ser redundantes o por haber sido reducidas). Una implementación directa del criterio de la cadena necesitaría mantener una lista de las parejas ya consideradas y la verificación del criterio tendría que realizar búsquedas sobre la lista. Una alternativa es reemplazar la lista por una matriz bidimensional indizada por una pareja de polinomios, en la que cada entrada almacena si la pareja correspondiente ha sido o no considerada. Una implementación que usa una matriz triangular de bits para este fin se presenta en [43].

Una implementación directa del criterio de la cadena presenta desventajas importantes. Por una parte, la lista de las parejas de polinomios ya consideradas debe actualizarse constantemente y puede llegar a consumir una cantidad significativa de memoria. Por otra parte, la

detección de redundancia depende de los elementos listados y esto a su vez depende del orden en el que los elementos se extraen de la cola de parejas pendientes. Si el orden de extracción de las parejas en la cola es arbitrario, también puede ocurrir que algunas parejas pendientes que se vuelven redundantes sólo son detectadas hasta ser extraídas, gastando memoria de forma innecesaria mientras permanecen en la cola. En [28], Gebauer y Möller implementaron el criterio de la cadena sin la necesidad de mantener una lista de parejas consideradas y purgando la cola de parejas pendientes sobre la marcha.

Algoritmo 2.1 Implementatión del criterio de la cadena (Gebauer y Möller)

función GEBAUER-MÖLLER($G, Q \subseteq B(x_1, x_2, \ldots, x_n), p \in G$)
  $C \leftarrow \{(p, g) : g \in G : g \neq p\}$
  para $p, g \in C : \neg criterio_1(p, g)$
    si $\exists p, g' \in C : g' \neq g \wedge mcm(lm(p), lm(g')) \mid mcm(lm(p), lm(g))$ **entonces**
      $C \leftarrow C \setminus (p, g)$
  $C \leftarrow C \setminus \{c \in C : criterio_1(c_1, c_2)\}$
  $Q \leftarrow Q \setminus \{q \in Q : lm(p) \mid mcm(lm(q_1), lm(q_2))$
    $\wedge\, mcm(lm(p), lm(q_1)) \neq mcm(lm(q_1), lm(q_2))$
    $\wedge\, mcm(lm(p), lm(q_2)) \neq mcm(lm(q_1), lm(q_2))\}$
  regresa $Q \cup C$

El algoritmo anterior se ejecuta cada vez que un polinomio p se agrega a la base. Inicialmente, la lista C de las parejas de polinomios que involucran a p incluye aquellas parejas que se pueden descartar por el criterio de coprimalidad. La razón de lo anterior es que, al momento de querer determinar si una pareja es descartable por el criterio de la cadena, el algoritmo necesita encontrar explícitamente en C a las parejas relacionadas y alguna de ellas podría cumplir el criterio de coprimalidad. En caso de que dos parejas c, c' cumplan el criterio de la cadena por relación mutua, sólo c se descarta de la lista. Esto se logra haciendo que la búsqueda de las parejas relacionadas con c' falle al haber descartado a c previamente. Debido a esto, importa el orden en el que las parejas se procesan. Las parejas que cumplen el criterio de coprimalidad se descartan posteriormente.

2.2. Implementaciones del algoritmo de Buchberger

Un polinomio booleano simplificado por las ecuaciones de campo es multilineal, es decir, ninguna variable tiene un exponente mayor que uno. Esto permite utilizar representaciones especializadas para ellos. En [34] se presenta BooleanGB, una implementación del algoritmo de Buchberger para calcular bases de Gröbner booleanas donde los monomios se representan

como arreglos de w bits. Este valor corresponde con el tamaño de un registro escalar del procesador, usualmente 32 o 64 bits. Bajo esta representación, $x_i(m)$ es igual al valor del i-ésimo bit en el arreglo, por lo que no es posible representar más de w variables en el monomio. La multiplicación monomial puede implementarse con una sola instrucción nativa del procesador, la cual corresponde con la disyunción bit a bit de dos arreglos de w bits cada uno y produce como resultado un monomio ya simplificado por las ecuaciones de campo. De manera similar, la división monomial es la disyunción exclusiva. Dado que $1m + 1m = 0m$ bajo aritmética en $\mathbb{Z}_2$, un polinomio booleano es visto como un conjunto de monomios y la suma de polinomios es su diferencia simétrica. BooleanGB está implementado en C++ y tiene una interfaz disponible en el sistema libre de álgebra computacional Macaulay2 [29].

Un polinomio booleano también es equivalente a una fórmula booleana escrita en forma normal algebraica y la función booleana correspondiente se puede representar de otras maneras. En [7] se presenta PolyBori, una implementación del algoritmo de Buchberger para calcular bases de Gröbner booleanas donde los polinomios se representan como diagramas de decisión con cero suprimido [39]. En estos diagramas, una función booleana se representa mediante un árbol binario tal que en la ramificación de cada nodo interno se decide el valor de verdad de alguna de las variables. Las hojas del árbol contienen la evaluación de la función booleana según las decisiones anteriores y las variables indefinidas al momento de llegar a la hoja toman el valor cero implícitamente. Puede ocurrir que un diagrama tenga tamaño exponencial en el número de variables de la función y que la suma en $\mathbb{Z}_2$ de dos funciones booleanas, vistas como diagramas de d_1 y d_2 nodos respectivamente, tome tiempo $\mathcal{O}(d_1 \cdot d_2)$ [36]. Los peores casos son poco frecuentes y PolyBori intenta reducir el tiempo de cómputo y el consumo de memoria mediante la memorización y compartición de subdiagramas que aparecen en distintos polinomios durante el cálculo. PolyBori está implementado en C++ y tiene una interfaz disponible en el sistema libre de álgebra computacional SageMath [16].

Otra implementación del algoritmo de Buchberger para calcular bases de Gröbner booleanas está disponible en el sistema propietario de álgebra computacional Magma [4]. Los detalles de su implementación no son públicos, pero se sabe que utiliza la estrategia de Gebauer y Möller en la implementación del criterio de la cadena, además de que algunos de sus parámetros son configurables y están documentados [6]. Dichos parámetros tienen que ver con la remoción de polinomios redundantes que provienen de la entrada antes de comenzar la generación de s-polinomios y con la interreducción de la base cuando un polinomio se agrega a la misma y éste reduce elementos previos. El algoritmo de Buchberger no es el algoritmo por omisión usado por Magma para el cálculo de bases de Gröbner booleanas.

2.3. El algoritmo F4 y técnicas de álgebra lineal

El algoritmo F4 para el cálculo de bases de Gröbner fue presentado por Faugére en 1999 y su aportación principal es el uso de técnicas de álgebra lineal para la aceleración masiva del proceso de reducción polinomial [22]. Las dos observaciones en las que está basado el algoritmo F4 son las siguientes. Por una parte, no queda claro cuál es la implementación óptima de la subrutina *selecciona* del algoritmo de Buchberger, por lo que una posibilidad es reducir al mismo tiempo un subconjunto (no necesariamente propio) de los elementos de la cola. Por otra parte, la eliminación gaussiana puede emular la reducción de múltiples polinomios si se aplica sobre una matriz que contenga, además de los polinomios a reducir, una fila por cada reductor que pudiera requerirse durante la reducción. Esto permite aprovechar lo que se conoce sobre las implementaciones de alto rendimiento de rutinas de álgebra lineal.

La construcción de la matriz es fundamental para la correctitud del algoritmo F4. En ésta, las filas denotan polinomios mientras que las columnas denotan monomios y las celdas contienen los coeficientes monomiales. La correspondencia de columnas con monomios se da según el orden monomial, de mayor a menor. Sea P el conjunto de polinomios a reducir y G la base actual, la matriz iniciamente contiene una fila por cada elemento de $P \cup G$. La reducción de $p \in P$ usando $g \in G$ puede implementarse como la resta de filas $p \leftarrow p - g$ sólo cuando $lm(p) = lm(g)$, lo que significa que el primer coeficiente diferente de cero de ambas filas aparece en la misma columna. Si $lm(g)$ divide estrictamente a $lm(p)$, debe existir en la matriz la fila $\frac{lm(p)}{lm(g)}g$ para poder emular la reducción usando sólo operaciones de filas. El preprocesamiento del algoritmo F4 examina qué monomios no son líderes en la matriz. Si alguna reducción pudiera requerir una fila con un monomio líder ausente pero la fila es generable como múltiplo monomial de una fila existente, entonces se genera dicha fila.

En el algoritmo F4, el conjunto P de polinomios a reducir no es un conjunto de s-polinomios como tal. Sea $sp(p,q) = \frac{\lambda_{p,q}}{lm(p)}p - \frac{\lambda_{p,q}}{lm(q)}q$ con $\lambda_{p,q} = mcm(lm(p), lm(q))$. Si el s-polinomio $sp(p,q)$ debe reducirse, entonces se agregan a P las parejas de productos no evaluados $(p, \frac{\lambda_{p,q}}{lm(p)})$ y $(q, -\frac{\lambda_{p,q}}{lm(q)})$. Esto permite detectar cuando s-polinomios diferentes tienen sumandos en común, pudiendo eliminar de la matriz las filas provenientes de productos no evaluados repetidos. A su vez, el producto no evaluado (q, m) expone a q como un divisor de $g \in G$ cuando $lm(g) = lm(q) \cdot m$, información que es de utilidad durante la fase de preprocesamiento del algoritmo F4. La publicación original también menciona una heurística que selecciona el reductor de una fila en caso de que hubieran varios reductores candidatos.

Faugére sugiere utilizar la estrategia grado-truncada para seleccionar el conjunto de parejas a extraer de la cola. Esta estrategia consiste en extraer todas las parejas cuyos s-polinomios

Algoritmo 2.2 Algoritmo F4

función F4($G \subseteq B(x_1, x_2, \ldots, x_n)$)
  $Q \leftarrow \{(p_i, p_j) : p_i, p_j \in G \wedge i < j\}$
  mientras $Q \neq \emptyset$
    $Q' \leftarrow \mathit{selecciona}(Q), Q \leftarrow Q \setminus Q'$
    $P \leftarrow \bigcup\{(q_1, \frac{\lambda_{q_1,q_2}}{lm(q_1)}), (q_2, -\frac{\lambda_{q_1,q_2}}{lm(q_2)}) : q \in Q' \wedge \neg \mathit{criterio}_1(q_1, q_2) \wedge \neg \mathit{criterio}_2(G, q_1, q_2)\}$
    $M \leftarrow \mathit{preprocesa}(P, G)$
    $R \leftarrow \mathit{reduce}(M) \setminus G$
    para cada $r \in R : r \neq 0$
      $Q \leftarrow Q \cup \{(r, g) : g \in G\}$
      $G \leftarrow G \cup r$
  regresa G

tengan grado menor o igual a cierto grado límite. El límite se incrementa si ningún s-polinomio pendiente se puede seleccionar bajo el límite actual. Cuando se usan órdenes monomiales graduados, limitar el grado de los s-polinomios a reducir sirve también para acotar la cantidad de monomios distintos (columnas de la matriz) que aparecen durante el cálculo. El algoritmo F4 con la estrategia grado-truncada es el algoritmo por omisión usado por Magma para el cálculo de bases de Gröbner booleanas.

La principal desventaja del algoritmo F4 es que la matriz generada en cada proceso de reducción puede ser excesivamente grande. Faugére sugiere usar una representación dispersa para las filas y aplicar compresión diferencial. En este esquema, se listan las posiciones con coeficientes diferentes de cero y se almacenan las diferencias entre posiciones listadas consecutivamente usando una cantidad de bits que depende de la magnitud de cada diferencia. En [27] se utiliza el esquema anterior cuando hay pocos coeficientes diferentes de cero y se utiliza una representación densa en el caso contrario, además de que se explora un método de compresión que opera sobre grupos de filas en lugar de comprimir cada una individualmente.

2.4. El algoritmo F5 y la estrategia basada en firmas

El algoritmo F5 para el cálculo de bases de Gröbner fue presentado por Faugére en 2002 y su aportación principal es el mejoramiento de los criterios de detección de reducciones redundantes [23]. En este algoritmo, cada uno de los polinomios está anotado con una firma, la cual contiene información parcial sobre cómo fue generado dicho polinomio (por ejemplo, si es un polinomio de la entrada o fue generado como una combinación lineal de otros polinomios). Durante la ejecución del algoritmo, las firmas de una pareja de polinomios son examinadas para intentar predecir si su s-polinomio reduce a cero. En el caso de sistemas homogéneos,

Faugére demostró que se detectan todas las reducciones redundantes.

Antes de considerar el k-ésimo polinomio de la entrada, el algoritmo F5 calcula la base de Gröbner de los $k-1$ polinomios anteriores y por esto se considera un algoritmo incremental. El conjunto de los s-polinomios detectados como redudantes por F5 no siempre es un superconjunto de los s-polinomios detectados por los criterios de Buchberger [19]. Además, la eficiencia del algoritmo suele no ser la esperada en el caso de sistemas no homogéneos [18]. La conversión de un sistema no homogéneo a uno homogéneo, la cual se logra mediante la introducción de variables artificiales, es muy común en la práctica. Sin embargo, el cálculo de bases de Gröbner para los sistemas extendidos a veces es más difícil que el cálculo sobre el sistema original [22]. El estudio de estrategias no incrementales además de la paralelización de los algoritmos basados en firmas es un área de investigación sumamente activa [17, 20].

2.5. El sistema criptográfico HFE y el Reto HFE 1

El sistema criptográfico HFE (del inglés *Hidden Field Equations*, que se traduce como "ecuaciones de campo escondidas") fue introducido en 1996 por Patarin [42]. La idea principal del sistema HFE es la siguiente. Sea $p \in \mathbb{Z}_{2^n}[x]$ restringido a $0 \leq x < 2^n$. Si el grado máximo de p está acotado por arriba por un entero d y d no es muy grande, entonces es posible encontrar rápidamente una de sus soluciones pues p tiene una única variable. A su vez, es posible transformar p en un sistema público de polinomios cuadráticos en $\mathbb{B}(x_1, x_2, \ldots, x_n)$ tal que la estructura original de p queda escondida y el sistema aparenta ser aleatorio. Como ya se ha comentado, resolver un sistema de este tipo es un problema $\mathcal{NP}$-Duro [1].

El Reto HFE 1 fue propuesto simultáneamente con el sistema criptográfico HFE y consiste en resolver un sistema de 80 ecuaciones binarias cuadráticas, generado a partir de un polinomio secreto con parámetros $n = 80$ y $d = 96$. El reto fue resuelto por primera vez por Faugére en 2002 utilizando una variante booleana de su algoritmo F5, la cual fue implementada en C y forma parte de la biblioteca de código cerrado FGb [24, 26]. El cálculo tomó 52 horas en una computadora Alpha EV68 corriendo a 1GHz. Al poco tiempo Magma también logró resolver el reto, reportando en 2004 que a su implementación del algoritmo F4 le tomó 22 horas completar el cálculo en una Sunfire v880 corriendo a 750MHz [44]. Un estudio reciente muestra que las implementaciones de Magma y FGb siguen siendo las más rápidas de su clase (algoritmos basados en álgebra lineal y basados en firmas respectivamente) [43].

En 2009 se reportó que el algoritmo MXL3 para el cálculo de bases de Gröbner también puede resolver el Reto HFE 1 [40]. Este algoritmo usa técnicas de álgebra lineal pero no se basa en el cálculo de s-polinomios. Su estrategia es un refinamiento del algoritmo XL, el cual

genera todos los múltiplos monomiales de los polinomios de la entrada hasta cierto grado d y luego interreduce el conjunto de polinomios, incrementando d hasta eventualmente encontrar una base de Gröbner [14]. La implementación del algoritmo MXL3 no está disponible públicamente, además de que el hardware y el tiempo de cálculo usados para resolver el reto no fueron reportados. Sólo se menciona que, a pesar de ser más lento que Magma 2.15, el algoritmo MXL3 genera una matriz que tiene menos filas que las matrices de FGb y Magma.

Capítulo 3

Contribuciones

Este capítulo está dividido en dos partes. En la primera, presentamos una implementación del algoritmo F4 para el cálculo de bases de Gröbner booleanas que presenta un rendimiento competitivo con el de las mejores implementaciones cerradas del estado de la técnica. En la segunda parte, presentamos dos algoritmos para calcular bases de Gröbner booleanas partiendo del conjunto de soluciones (o no soluciones) del sistema en cuestión. Tales algoritmos no están basados en la teoría de Buchberger y, para sistemas con pocas variables, son más rápidos aún incluyendo el tiempo que toma calcular todas las soluciones por fuerza bruta. Hemos publicado parte del trabajo descrito en este capítulo en [10] y [11].

3.1. Variante booleana del algoritmo F4

En esta sección presentamos los detalles algorítmicos y de implementación de nuestra variante del algoritmo F4 para el cálculo de bases de Gröbner booleanas. Comenzamos dando una descripción general de nuestro algoritmo y listamos una serie de suposiciones que determinan características importantes de la implementación. Posteriormente describimos cómo manejamos la cola de parejas pendientes y describimos nuestra implementación concurrente de los criterios de Buchberger. Más adelante introducimos un esquema de compresión para polinomios de sólo lectura que reduce el consumo de memoria de nuestro algoritmo en una amplia variedad de situaciones. Después describimos nuestra implementación de la etapa de preprocesamiento y de nuestro proceso concurrente de reducción polinomial, a la vez que explicamos cómo ajustar sus parámetros de configuración para lograr un óptimo rendimiento sobre la arquitectura del procesador usada para el cálculo. Por último discutimos el criterio de terminación de nuestro algoritmo, el cual permite ocasionalmente determinar que la base ya es de Gröbner, antes de tener que realizar trabajo adicional.

3.1.1. Descripción general y suposiciones

Como la mayoría de las implementaciones del algoritmo F4, nuestra variante utiliza la estrategia grado-truncada sugerida por Faugére [22]. Nuestra implementación es compatible con cualquier orden monomial graduado pero por el momento sólo cuenta con el orden graduado lexicográfico reverso, que también es el orden más popular en la práctica. Al igual que en BooleanGB, un monomio booleano se representa utilizando un arreglo de bits que denotan los exponentes de las variables en el monomio. Por otra parte, nuestra implementación no está limitada a utilizar arreglos de 32 o 64 bits como en BooleanGB, sino que puede utilizar arreglos con tamaño suficiente para representar todas las variables del sistema.

Definimos la posición de un monomio en el orden monomial dado como la cantidad de monomios menores que él. Un polinomio booleano se representa como una secuencia de enteros no negativos que denotan las posiciones de los monomios que aparecen en el polinomio. Dos algoritmos realizan la conversión (en ambos sentidos) entre un monomio visto como arreglo de bits y la posición que le corresponde en el orden monomial.

Algoritmo 3.1 Monomio a posición (orden graduado lexicográfico reverso, n variables)

función POSICIÓN($m \in \mathbb{B}(x_1, x_2, \ldots, x_n)$)
  $k \leftarrow 0, d \leftarrow 0$
  para $i \leftarrow 1 \ldots n : x_i(m) = 1$
    $d \leftarrow d + 1$
    $k \leftarrow k + \binom{i-1}{d}$
  regresa $\sum_{i=0}^{deg(m)} \binom{n}{i} - k - 1$

Algoritmo 3.2 Posición a monomio (orden graduado lexicográfico reverso, n variables)

función MONOMIO($k, n \in \mathbb{N}$)
  $m \leftarrow \mathbb{B}(x_1, x_2, \ldots, x_n)\{1\}$
  $d \leftarrow \text{mín}\,\{d' \in \mathbb{N} : k < \sum_{i=0}^{d'} \binom{n}{i}\}$
  $k \leftarrow \sum_{i=0}^{d} \binom{n}{i} - k - 1$
  mientras $k \geq 0$
    $n \leftarrow n - 1$
    si $k \geq \binom{n}{d}$ **entonces**
      $x_{n+1}(m) \leftarrow 1$
      $k \leftarrow k - \binom{n}{d}$
      $d \leftarrow d - 1$
  para cada $i \in [1, d]$
    $x_i(m) \leftarrow 1$
  regresa m

El coeficiente binomial $\binom{n}{k}$ puede calcularse y memorizarse con técnicas de programación dinámica. A su vez, nuestra implementación memoriza los monomios de las distintas posiciones monomiales que existen bajo el grado límite de la estrategia grado-truncada. Por otra parte, existen varias razones por las que utilizamos enteros de 32 bits sin signo para representar posiciones en el orden monomial y para realizar operaciones aritméticas, pese a que el número de monomios distintos crece exponencialmente con el número de variables. Sea d el grado límite de la estrategia grado-truncada en algún punto de la ejecución, el número de columnas en la matriz está acotado por arriba por $\sum_{i=0}^{d} \binom{n}{d}$. En la práctica, el número de filas es de una magnitud comparable. Con las limitaciones tecnológicas actuales en tiempo y memoria, no se considera factible procesar matrices con más de 2^{32} filas y columnas. En perspectiva, el cálculo de una base de Gröbner booleana para un sistema con $n = 100$ variables debe terminar en un grado máximo $d = 6$ para evitar problemas de desbordamiento de enteros de 32 bits y podrían requerirse petabytes de memoria para el cálculo de la base. Modificar la implementación para usar enteros de 64 bits es viable, pero el aumento en el tiempo de cálculo y en el consumo de memoria sería apreciable.

En nuestra implementación, el grado inicial de la estrategia grado-truncada es 2 y se asume que es el grado máximo de los polinomios de entrada. Los polinomios de la base se almacenan en un arreglo indizado por la posición de su monomio líder y tiene una capacidad igual al número de monomios distintos que existen bajo el grado límite actual, por lo que no recomendamos incrementar el grado inicial a no ser que exista alguna justificación mayor.

3.1.2. Manejo de la cola de parejas pendientes

La estrategia grado-truncada restringe qué parejas generadoras de s-polinomios pueden extraerse de la cola, pero no restringe las parejas que pueden agregarse a la misma. Cuando un s-polinomio no redundante de grado d aparece, las implementaciones del estado de la técnica agregan por omisión la pareja generadora a la cola. Sin embargo, si el cómputo logra encontrar una base de Gröbner en un grado límite menor que d, haberla agregado a la cola fue innecesario en retrospectiva. Para reducir el consumo de memoria de la cola, Magma provee un parámetro que restringe estáticamente el grado de los s-polinomios cuyas parejas pueden agregarse a la cola [6]. Durante la ejecución se asume que ningún s-polinomio de grado mayor será necesario para calcular correctamente la base.

Nuestra implementación restringe dinámicamente el grado de los s-polinomios cuyas parejas pueden agregarse a la cola sin condicionar la correctitud del cálculo de la base de Gröbner, lo cual mantiene el bajo consumo de memoria para la cola a expensas de realizar una pequeña cantidad de trabajo repetido. Nosotros usamos el mismo grado límite de la estrategia

grado-truncada para restringir tanto extracciones como inserciones en la cola. Cuando el grado límite se incrementa, todas las parejas se reexaminan en paralelo para insertar aquéllas que fueron ignoradas en etapas previas pero que se permiten con el nuevo límite.

Nuestra variante no almacena como tal las parejas no redundantes, sino que almacena los dos productos no evaluados $(p, \frac{\lambda_{p,q}}{lm(p)}), (q, \frac{\lambda_{p,q}}{lm(q)})$ que corresponden con los dos sumandos del s-polinomio generado por la pareja en cuestión. Note que $\frac{\lambda_{p,q}}{lm(q)} = -\frac{\lambda_{p,q}}{lm(q)}$ en el caso booleano. Para s-polinomios generados por un elemento de la base y una ecuación de campo, se almacenan en la cola todos los productos no evaluados (p, x_i) donde $x_i(p) = 1$ para $1 \leq i \leq n$, aunque sólo se insertan en la cola cuando el grado límite d es estrictamente mayor que $deg(p)$.

3.1.3. Implementación concurrente del criterio de la cadena

Aunque la instalación de Gebauer y Möller es ampliamente considerada como la mejor implementación del criterio de la cadena, tiene algunas desventajas. Por una parte, se sabe que cuando dos parejas pendientes c, c' cumplen el criterio de la cadena por relación mutua, una de ellas se puede descartar siempre y cuando la otra se procese. El algoritmo de Gebauer y Möller determina cuál descartar decidiendo secuencialmente para cada una de ellas: si se descarta c gracias a que c' existe, c se elimina de la lista de pendientes y no es visible cuando se tome la decisión sobre c'. Por otra parte, el algoritmo necesita que las parejas relacionadas que se utilicen durante las decisiones estén explícitamente listadas en la lista de parejas pendientes. Esto hace necesario construir (al menos temporalmente) parejas redundantes que cumplen el criterio de coprimalidad para que sean visibles durante la detección del criterio de la cadena. Nuestra implementación elimina la sobrecarga de listar explícitamente parejas redundantes así como la secuencialidad en la toma de decisiones.

Para lograr lo anterior, a cada polinomio p que entra a la base se le asigna un identificador único creciente $id(p)$. Dicho identificador es útil para generar un orden $<_{id}$ entre parejas de polinomios tal que $(p, q) <_{id} (p', q')$ si y sólo si $id(p) < id(p') \vee (id(p) = id(p') \wedge id(q) < id(q'))$. Para que el orden $<_{id}$ se asemeje al orden en el que una implementación secuencial tomaría las decisiones, se deben listar las componentes de las parejas de modo que $id(p) > id(q)$ e $id(p') > id(q')$. Esto equivale a que la pareja que involucre al polinomio más recientemente agregado a la base sea la mayor en el orden. Una idea inicial es descartar una pareja cuando sea posible encontrar dos parejas relacionadas que sean menores bajo $<_{id}$. Sin embargo, conviene definir un nuevo orden $<_*$ de parejas que utilice criterios más fuertes y sólo desempate usando $<_{id}$ en caso de ser necesario. Dicho orden además puede ser parcial, pues basta se pueda comparar la pareja (p, q) con sus dos parejas relacionadas $(p, g), (q, g)$ donde $lm(g) \mid \lambda_{p,q}$.

Es importante notar que si $lm(g) \mid \lambda_{p,q}$, entonces $deg(\lambda_{p,g}), deg(\lambda_{q,g}) \leq deg(\lambda_{p,q})$. Cuando

la desigualdad es estricta para alguna de las dos parejas, conviene definir $<_*$ de tal modo de que dicha pareja sea menor que (p, q) independientemente de la evaluación de $<_{id}$. Por razones similares, si una pareja relacionada es redundante por el criterio de coprimalidad, entonces conviene que sea menor bajo $<_*$ independientemente de la evaluación de $<_{id}$. Un tercer caso especial ocurre cuando $g \mid q$ y $q \mid p$, lo que implica que $g \mid p$. Preferimos descartar (p, q) a expensas de tener que reducir las parejas (p, g) y (q, g). Dado que $\lambda_{p,g} = \lambda_{p,q} = lm(p)$, nuestro algoritmo considera que (p, g) es menor que (p, q) bajo $<_*$ por el hecho de que $g \mid q$.

La siguiente implementación del criterio de la cadena asume que la base actual G no se modifica durante la toma de decisiones, pero no necesita conocer qué otras parejas están siendo examinadas concurrentemente. La implementación construye tuplas en las que la comparación lexicográfica simula $<_*$ seguido de $<_{id}$ para resolver empates.

Algoritmo 3.3 Implementación concurrente del criterio de la cadena

función CRITERIO$_2(G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n), p, q \in G)$
  $t_{p,q} \leftarrow (verdadero, deg(\lambda_{p,q}), verdadero, ordena_{>}(id(p), id(q)))$
  para cada $g \in G : g \neq p \wedge g \neq q \wedge lm(g) \mid \lambda_{p,q}$
    $t_{p,g} \leftarrow (\neg criterio_1(p, g), deg(\lambda_{p,g}), \neg(g \mid q), \max(id(p), id(g)), \min(id(p), id(g)))$
    $t_{q,g} \leftarrow (\neg criterio_1(q, g), deg(\lambda_{q,g}), \neg(g \mid p), \max(id(q), id(g)), \min(id(q), id(g)))$
    si $t_{p,q} > t_{p,g} \wedge t_{p,q} > t_{q,g}$ **entonces regresa** *verdadero*
  regresa *falso*

3.1.4. Compresión de polinomios de sólo lectura

Un polinomio p de la base permanece sin modificaciones hasta que aparece un polinomio q que pueda reducirlo. Cuando esto ocurre, p puede eliminarse de la base y sustituirse por su reducción. La reducción puede calcularse mediante el escalonamiento de una matriz que incluya tanto a p como a $\frac{lm(p)}{lm(q)}q$. Un polinomio reducible puede modificarse múltiples veces durante el escalonamiento de la matriz y nuestra implementación lo representa mediante un arreglo de 2^d bits, donde d es el grado del polinomio y cada bit denota la presencia o ausencia de los monomios según sus posiciones en el orden monomial. Para polinomios que aún son irreducibles, utilizamos una representación distinta y que consume menos memoria.

Un polinomio booleano puede verse como una secuencia de enteros no negativos ordenados y sin duplicados, los cuales denotan los monomios que aparecen en el polinomio según el orden monomial. Faugére sugiere almacenar las secuencias utilizando compresión diferencial. En ésta, se almacena el entero inicial de la secuencia y una lista de las diferencias entre enteros consecutivos de la secuencia utilizando tan pocos bits como sea posible para cada

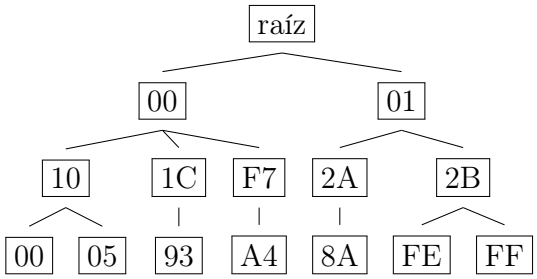


Figura 3.1: Ejemplo de un árbol de prefijos para un conjunto de siete enteros de 24 bits cada uno (bytes en hexadecimal). Un nodo puede tener a lo mucho 256 hijos.

diferencia. Nosotros utilizamos árboles de prefijos de bytes, los cuales son árboles enraizados que tienen una hoja por cada entero insertado. El camino de un entero tiene una longitud igual a la cantidad de bytes de la representación binaria del entero y el i-ésimo nodo a partir de la raíz almacena el i-ésimo byte más significativo. Si dos o más enteros comienzan con la misma secuencia de bytes, el subcamino correspondiente se comparte. Ver figura 3.1.

Aunque la compartición de subcaminos desde la raíz tiene el efecto buscado de comprimir la representación de la secuencia de enteros, una implementación cruda de árboles de prefijos almacena en cada nodo interno los apuntadores hacia sus hijos, lo que representa una sobrecarga significativa en el consumo de memoria. Dichos apuntadores permiten navegar el árbol libremente y también permiten modificarlo dinámicamente en caso de ser necesario. Sin embargo, nosotros utilizamos los árboles de prefijos para representar polinomios irreducibles y las operaciones que se realizan sobre ellos se limitan a iterar sobre la secuencia completa de enteros, ya sea para sumar el polinomio con otro que está siendo reducido o para migrar el polinomio a un arreglo de bits cuando aparezca un polinomio que pueda reducirlo. También puede ser útil conocer el monomio líder del polinomio, pero éste puede memorizarse de antemano. Bajo estas suposiciones, los apuntadores pueden obviarse.

Para lograr eliminar los apuntadores, el contenido de los nodos del árbol se almacena en preorden (primero el nodo padre y luego los hijos de izquierda a derecha). Lo anterior implica que para visitar un subárbol derecho, primero se debe avanzar sobre toda la representación binaria de los subárboles hermanos que estén a la izquierda. Esto es de poca relevancia si se iterará sobre la secuencia completa de todos modos. En cada nodo interno se almacena información que permite listar cada uno de los bytes correspondientes a los hijos inmediatos. La representación de un entero dado está completa cuando se llega a una hoja y los enteros pueden visitarse en orden si se construye el árbol de la manera adecuada. Ver figura 3.2.

Sea h la cantidad de hijos inmediatos de un nodo interno, nuestra implementación utiliza incondicionalmente cinco bits por nodo interno para almacenar el valor de $\text{mín}(h, 32)$ mód 32.

02, 00, 01	**03**, 10, 1C, F7	**02**, 00, 05	**01**, 93
01, A4	**02**, 2A, 2B	**01**, 8A	**02**, FE, FF

Figura 3.2: Ejemplo de serialización en preorden del árbol anterior. Cada caja representa el contenido de un nodo interno. El número de hijos de cada nodo se muestra en negritas.

Si $h < 32$, entonces se utiliza una lista de h bytes para almacenar los octetos de los hijos inmediatos. Si $h \geq 32$, entonces el preludio de cinco bits almacena un 0 y se utiliza un arreglo de 256 bits para almacenar la presencia o ausencia de los octetos correspondientes a los hijos inmediatos. Nuestra implementación almacena cada tipo de secuencia (los preludios, las listas de octetos y los arreglos de bits) en secuencias separadas. Por una parte, esto permite simplificar la navegación sobre las representaciones binarias, pues la secuencia de preludios es la única que no puede procesarse con granularidad de bytes. Por otra parte, la secuencia de arreglos de bits podría manipularse con instrucciones vectoriales del procesador, las cuales son capaces de operar sobre múltiples palabras a la vez, siempre y cuando dicha secuencia esté correctamente alineada en memoria (debe comenzar en una dirección de memoria múltiplo del tamaño en bytes de los registros vectoriales a usar, usualmente 16 o 32 bytes).

Algoritmo 3.4 Compresión de polinomio (ejemplo de árbol con tres octetos por entero)

función COMPRESIÓN($p \in \mathbb{B}(x_1, x_2, \ldots, x_n) \setminus 0, s \in \mathbb{Z}_2^*$)
  $u \leftarrow \mathbb{Z}_2^{256}\{0, 0, \ldots, 0\}, v \leftarrow \mathbb{Z}_2^{256}\{0, 0, \ldots, 0\}, w \leftarrow \mathbb{Z}_2^{256}\{0, 0, \ldots, 0\}$
  para $i \leftarrow$ *posición*$(lm(p)) \ldots 0$
    $u_{i \bmod 2^8} \leftarrow$ *coef*$(p_{monomio(i)})$
    si $i \equiv 0 \pmod{2^8}$ **entonces**
      si $|u| \neq 0$ **entonces**
        $s \leftarrow$ *concatena*$(s,$ *invierte*$($*representación*$(u)))$
        $u \leftarrow \{0, \ldots, 0\}$
        $v_{\frac{i}{2^8} \bmod 2^8} \leftarrow 1$
      si $i \equiv 0 \pmod{2^{16}}$ **entonces**
        si $|v| \neq 0$ **entonces**
          $s \leftarrow$ *concatena*$(s,$ *invierte*$($*representación*$(v)))$
          $v \leftarrow \{0, \ldots, 0\}$
          $w_{\frac{i}{2^{16}} \bmod 2^8} \leftarrow 1$
  $s \leftarrow$ *concatena*$(s,$ *invierte*$($*representación*$(w)))$
  regresa *invierte*(s)

El algoritmo que toma un polinomio representado como un arreglo de bits y produce la serialización sin apuntadores de su árbol de prefijos no necesita construir como paso intermedio el árbol con apuntadores. Nuestro algoritmo simula el recorrido opuesto al recorrido

en preorden directamente sobre el arreglo de bits del polinomio no comprimido, serializando el árbol sobre un bloque de memoria temporal lo suficientemente grande como para poder almacenar la serialización de cualquier polinomio con el mismo monomio líder. Al terminar el recorrido, la representación binaria de la serialización fue escrita en la memoria temporal en el orden inverso al que debe tener. Finalmente, se aparta un bloque de memoria permanente con un tamaño mínimo necesario y se copia la salida del paso anterior en sentido inverso.

3.1.5. Preprocesamiento previo a la reducción polinomial

En el algoritmo F4, la etapa de preprocesamiento es la encargada de construir la matriz que será escalonada para simular el proceso de reducción polinomial. Esta matriz incluye filas provenientes de parejas no redundantes, filas que denotan elementos de la base y también los múltiplos monomiales de elementos de la base que sean necesarios para simular la reducción polinomial correctamente mediante cualquier implementación eficiente de la eliminación gaussiana. Sin embargo, nuestra implementación del proceso de reducción es específica para matrices provenientes de cálculos de bases de Gröbner, en parte porque prácticamente todas las filas de las matrices son múltiplos monomiales de los elementos de la base. A continuación describimos la implementación de nuestra etapa de preprocesamiento, para posteriormente describir en la siguiente subsección la implementación de nuestro proceso de reducción.

Dado que nuestra implementación restringe tanto extracciones como inserciones a la cola de parejas pendientes, en principio consideramos la totalidad de los elementos en la cola para llevar a cabo el preprocesamiento. Los productos no evaluados provenientes de las parejas pendientes se ordenan y los productos repetidos se eliminan. También eliminamos los productos no evaluados de la forma $(p, 1)$ que surgieron de parejas polinomiales (p, q) cuando $q \mid p$, bajo el argumento de que p ya es una fila de la base. Eventualmente, los productos que no pudieron ser eliminados se evaluarán y se añadirán a la matriz que se llevará a su forma escalonada, dando como resultado la reducción de los s-polinomios correspondientes.

Sean $p, q \in G$ donde $lm(q) \mid lm(p)$, la fila reductora de p construida usando q es el múltiplo monomial $r = \frac{lm(p)}{lm(q)}q$. Dado que el $lm(r) = lm(p)$, el primer coeficiente distinto de cero en ambas filas de la matriz está en la misma columna y la reducción de p puede realizarse sumando r. En esta etapa, nosotros buscamos una manera de generar una única fila reductora para cada monomio que aparezca bajo el grado límite actual y preferimos usar algún producto no evaluado que de todos modos deba generarse. Si para algún monomio m existe algún producto no evaluado $(p, \frac{m}{lm(p)})$, entonces la fila reductora de m la generaremos eligiendo el producto con el menor $lm(p)$. En caso contrario, elegimos el $g \in G$ con el menor $lm(g)$ tal que $lm(g) \mid m$. La decisión de elegir el menor reductor disponible es simplemente una heurística.

Desafortunadamente, las elección de reductores influye en qué monomios líderes aparecerán en la matriz escalonada y es difícil determinar cuáles son las elecciones más favorables. La publicación original del algoritmo F4 menciona una heurística cuyo objetivo es diversificar la elección de reductores de un escalonamiento al siguiente. Sólo hemos encontrado un reporte conciso sobre el efecto de aplicar dicha heurística y se encontró que el cálculo se acelera ocasionalmente a expensas de utilizar una considerable cantidad de memoria adicional [41].

La diferencia más importante de nuestra variante con respecto al algoritmo F4 es que nuestra etapa de preprocesamiento no genera anticipadamente las filas reductoras, sino que únicamente anota qué polinomio de la base será el que genere el reductor de cada monomio líder. Como se explicará posteriormente, nuestra implementación construye las filas reductoras bajo demanda durante el escalonamiento de la matriz. Más aún, los productos no evaluados que se elijan para generar las filas reductoras también se eliminan de la cola de pendientes bajo el siguiente argumento. Existen dos tipos de productos no evaluados: los de forma (p, x_i) donde $x_i(p) = 1$, provienentes de s-polinomios con ecuaciones de campo y los de forma $(p, \frac{\lambda_{p,q}}{lm(p)})$, provienentes de s-polinomios entre elementos explícitos de la base. Cuando un producto no evaluado (p, f) cumple que $lm(p)$ y f son coprimos, entonces es un producto del segundo tipo donde $(p, f) = (p, \frac{\lambda_{p,q}}{lm(p)})$ con $q \in G$. Además, el producto $(q, \frac{\lambda_{p,q}}{lm(q)})$ también aparece en la cola y ambos pueden generar una fila reductora para $m = \lambda_{p,q}$. Si generamos la fila $p \cdot f$ bajo demanda cada vez que un polinomio con monomio líder $\lambda_{p,q}$ deba reducirse, entonces (p, f) puede eliminarse de la cola, siempre y cuando $(q, \frac{\lambda_{p,q}}{lm(q)})$ sí se procese. Como eliminamos a lo mucho un producto de la cola por cada monomio líder, $sp(p, q)$ se reducirá correctamente. Cabe mencionar que interreducimos los polinomios de entrada antes de agregarlos a la base, evitando así tener polinomios con monomios líderes repetidos.

Algoritmo 3.5 Preprocesamiento de la variante de F4

función PREPROCESAMIENTO($G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n), P \subseteq (G \times x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}), d \in \mathbb{N}$)
  $P \leftarrow P \setminus \{p \in P : p_2 = 1\}$
  $R \leftarrow \mathbb{B}(x_1, x_2, \ldots, x_n)^{\sum_{i=0}^{d} \binom{n}{i}} \{0, 0, \ldots, 0\}$
  para $i \leftarrow \sum_{i=0}^{d} \binom{n}{i} - 1 \ldots 0$
    $C_P \leftarrow \{p \in P : p_2 = \frac{monomio(i)}{lm(p_1)}\}$
    $C_G \leftarrow \{g \in G : lm(g) \mid monomio(i)\}$
    si $|C_P| \neq 0$ **entonces**
      $R_i \leftarrow (\text{argmin}_{C_P}\, lm(p_1))_1$
      $P \leftarrow P \setminus \text{argmin}_{C_P}\, lm(p_1)$
    si no si $|C_G| \neq 0$ **entonces**
      $R_i \leftarrow \text{argmin}_{C_G}\, lm(g)$
  regresa (P, R)

3.1.6. Implementación del proceso de reducción

Esta etapa es la encargada de reducir los polinomios denotados por los productos no evaluados que permanecieron después del preprocesamiento y también es la que consume la mayor parte del tiempo total del cálculo de la base de Gröbner. Existen muchas implementaciones posibles y nosotros consideramos que la nuestra es un híbrido entre la reducción al estilo del algoritmo Buchberger y la del algoritmo F4. Por una parte, el algoritmo de Buchberger reduce individualmente cada polinomio p hasta que $p = 0$ o $lm(p)$ sea irreducible. Sea p un polinomio reducible y $R_i \in \mathbb{B}(x_1, x_2, \ldots, x_n)$ con $i = posición(lm(p))$ el polinomio elegido para reducir los polinomios con monomio líder $lm(p)$, el polinomio $r = \frac{lm(p)}{lm(R_i)} R_i$ se construye únicamente para reducir p mediante la suma $p + r$ y posteriormente se desecha. Por otra parte, el algoritmo F4 construye una matriz que incluye explícita y anticipadamente todos los polinomios r que pudieran ser necesarios además de todos los productos pendientes ya evaluados y luego escalona la matriz. La ventaja del algoritmo de Buchberger es su bajo consumo de memoria al mantener sólo un polinomio reductor r activo a la vez, aunque esto implique tener que volver a construir el mismo r si la reducción de otro polinomio lo llegara a requerir después. La ventaja del algoritmo F4 es el no tener que construir repetidamente las filas reductoras r y poder delegar la reducción al escalonamiento de una matriz, lo cual puede ser bastante rápido a expensas de ocupar mucha memoria. Nuestra implementación de la etapa de reducción busca conservar el bajo consumo de memoria del algoritmo de Buchberger, conservando la mayor parte del aceleramiento del algoritmo F4.

Sea P el conjunto de polinomios a reducir. El problema de reducir P teniendo sólo un reductor activo al mismo tiempo y generando cada reductor a lo mucho una vez, se puede resolver de la siguiente manera. Sea $m = \text{máx}\{lm(p) : p \in P\}$ con $i = posición(m)$. Si $R_i = 0$ entonces se elige algún $p \in P$ tal que $m = lm(p)$ y éste se asigna como el nuevo valor de R_i, se agrega a la base y también se elimina de P. Si $R_i \neq 0$ entonces se genera $r = \frac{m}{lm(R_i)} R_i$ y todos los $p \in P$ tales que $m = lm(p)$ se reducen concurrentemente sobre él. Después de realizar todas las reducciones sobre r, se tiene que $\text{máx}\{lm(p) : p \in P\} < m$ y r puede desecharse. El proceso se repite hasta que todos los polinomios de P reduzcan a cero. Dado que $lm(p) = lm(r)$, la reducción de p sobre r es equivalente a $p + r$ y también a $p \oplus r$ si ambos polinomios booleanos están representados como arreglos de bits. Ya que los polinomios de P serán modificados frecuentemente durante la reducción, nuestra implementación ya los representa de esta manera. Al momento de generar r, nuestra implementación también utiliza un arreglo de bits para representarlo y además lista las posiciones de sus subarreglos disjuntos de 256 bits que tienen bits prendidos. Cuando necesitamos calcular $p \oplus r$, estimamos si conviene visitar r completo o sólo los subarreglos de r con bits prendidos. Para minimizar

la cantidad de accesos a memoria, nuestra implementación hace lo segundo si la memoria que ocupa la lista de posiciones más la memoria que ocupan los subarreglos indicados en dicha lista es menor que la memoria total de r. Además, la disyunción exclusiva entre dos subarreglos de 256 bits la llevamos a cabo con instrucciones del procesador que operan sobre registros vectoriales de 128 o 256 bits, dependiendo del tamaño máximo disponible.

Desafortunadamente, el algoritmo anterior es sumamente lento en la práctica por la multitud de accesos a memoria principal que realiza. Para explicar por qué el algoritmo realiza una cantidad excesiva de accesos a memoria principal, así como cuál es la sobrecarga correspondiente, debemos explicar algunos aspectos importantes del hardware y en particular de la jerarquía de memoria de los procesadores actuales. En un procesador, el tiempo que toma la ejecución de una instrucción que opera sobre registros se mide en ciclos y si el procesador opera a 3 gigahertz, entonces ocurren tres mil millones de ciclos en un segundo, lo que equivale a $\frac{1}{3}$ de nanosegundo por ciclo. Sin embargo, los tiempos de acceso a memoria tienden a medirse en nanosegundos debido a que su velocidad se ha estancado en comparación con la frecuencia del procesador. Mientras que los registros del procesador sí operan a la frecuencia de éste, la memoria principal tiene tiempos de acceso de entre 50 y 100 nanosegundos, lo que representa cientos de ciclos del procesador [32]. Para ocultar esta sobrecarga, algunos procesadores incluyen diferentes niveles de cachés, donde las cachés más rápidas son también las más pequeñas por cuestiones de costo y consumo de energía. Por ejemplo, en los procesadores de Intel es común encontrar tres niveles de caché: la caché nivel 1 (L1) con tiempos de acceso de 1 nanosegundo pero limitada a decenas de kilobytes, la caché nivel 2 (L2) con tiempos de acceso de entre 3 y 10 nanosegundos pero limitada a centenas de kilobytes, y la cache nivel 3 (L3) con tiempos de entre 10 y 20 nanosegundos pero limitada a una decena de megabytes [31, 32]. En procesadores con múltiples núcleos, la caché L3 generalmente se comparte entre todos los núcleos mientras que las cachés de menor nivel son propias de cada núcleo físico. Acceder a un dato que no está en la caché implica leerlo de la memoria principal y en general, el dato también se almacenará en la caché y permanecerá en la misma hasta que otros datos logren sacarlo. Si se vuelve a usar un dato que está en la caché, no es necesario ir a la memoria principal y el tiempo de permanencia del dato en la caché se alarga.

En el algoritmo anterior, todos los polinomios de P se reducen concurrentemente sobre el único reductor r y es fácil ver que r permanecerá en la caché al usarse constantemente por todos los núcleos del procesador (probablemente estará en la caché L2 de cada núcleo y seguramente estará en la caché compartida L3). Por otra parte, si cada $p \in P$ ocupa decenas o cientos de kilobytes de memoria, sólo cabrán algunas decenas de ellos incluso en la caché más grande. Si P tiene miles de polinomios y cada uno se lee sólo para aplicar su reducción

sobre r, cuando la reducciones de todos sobre r terminen y ahora P deba reducirse sobre el siguiente reductor r', prácticamente la totalidad de los polinomios de P tendrán que volver a leerse de la memoria principal. Una manera de acelerar masivamente el algoritmo anterior es la siguiente: en lugar de tener activo un único reductor r, podemos tener un conjunto de reductores activos T para que cada vez que un $p \in P$ deba leerse de la memoria principal, al menos se pueda reducir múltiples veces antes de que salga de la caché. El número de reductores activos $|T|$ debe ser pequeño para no aumentar el consumo de memoria al estilo del algoritmo F4 y también debe ser lo suficientemente grande como para evitar que las cachés se subutilicen. Suponiendo que el procesador tiene L núcleos lógicos y que a cada uno se le asigna un polinomio de P por reducir, nuestra implementación procura que la memoria total de los $|T| + L$ polinomios que se están leyendo o modificando concurrentemente sea cercana al tamaño de la caché L3. Fuera del hecho de minimizar el número de accesos realizados a la caché L3 durante el cálculo de $p \oplus r$ al tomar en cuenta las posiciones de los bits prendidos de r, nuestra implementación no trata de manera especial a las cachés L2 o L1.

Algoritmo 3.6 Reducción polinomial en la variante del algoritmo F4

función REDUCCIÓN($P, G \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n), d, k \in \mathbb{N}, R \in \mathbb{B}(x_1, x_2, \ldots, x_n)^{\sum_{i=0}^{d} \binom{n}{i}}$)
  $v_f \leftarrow \sum_{i=0}^{d} \binom{n}{i}$
  mientras $v_f \neq 0$
    $v_i \leftarrow v_f - \min(v_f, k)$
    $T \leftarrow \mathbb{B}(x_1, x_2, \ldots, x_n)^k\{0, 0, \ldots, 0\}$
    para cada $i \in [v_i, v_f)$
      si $R_i \neq 0$ **entonces**
        $T_{i-v_i+1} \leftarrow \frac{monomio(i)}{lm(R_i)} \cdot R_i$
    para cada $p \in P$
      mientras $p \neq 0 \wedge v_i \leq posición(lm(p)) < v_f$
        si $T_{posición(lm(p))-v_i+1} = 0$ **entonces**
          $T_{posición(lm(p))-v_i+1} \leftarrow p$
          $G \leftarrow G \cup p$
          $P \leftarrow P \setminus p$
          termina ciclo interno
        si no
          $p \leftarrow p + T_{posición(lm(p))-v_i+1}$
    $v_f \leftarrow v_i$
  regresa G

Una condición de carrera ocurre cuando dos o más hilos de ejecución acceden simultáneamente a la misma celda de memoria y al menos uno de ellos realiza una operación de

escritura. Si dos hilos reducen concurrentemente los polinomios p_1, p_2 donde $lm(p_1) = lm(p_2)$ y el reductor para tal monomio líder no existe, ambos hilos intentarán establecer su polinomio como la nueva fila pivote correspondiente al monomio líder. Para conservar la correctitud de la implementación, sólo uno de los hilos debe poder asignar su polinomio como la nueva fila y el otro deberá reducir su polinomio sobre ésta última. Nuestra implementación evita condiciones de carrera mediante el uso de instrucciones atómicas del procesador. En particular, la pregunta sobre si $T_{posición(lm(p))-v_i+1} = 0$ seguida en caso afirmativo de la asignación sobre él mismo, puede implementarse con una instrucción atómica de comparación con intercambio.

El resultado de evaluar un producto pendiente es un polinomio que debe reducirse y éste se almacena en un arreglo de bits al igual que los reductores activos; los primeros porque serán modificados y los segundos porque son pocos y dicha representación facilita una implementación eficiente del proceso de reducción $p \oplus r$. Sin embargo, si el número de productos pendientes es muy grande entonces la cantidad de memoria utilizada puede ser considerable. Para disminuir el consumo de memoria, nuestra implementación parte el conjunto de productos pendientes en subconjuntos disjuntos de a lo mucho M elementos, y luego evalúa y reduce cada subconjunto secuencialmente. Después de reducir un subconjunto, los polinomios que se agreguen a la base se almacenan comprimidos, por lo que el consumo máximo de memoria es menor en comparación con evaluar todos los productos pendientes desde el inicio. La desventaja es que cada reductor r ya no se generará una única vez, sino una vez por cada subconjunto de polinomios pendientes que se reduzca de manera independiente.

3.1.7. Reinicialización y terminación del algoritmo

Durante el cálculo de la base de Gröbner pueden aparecer nuevos polinomios constantes, lineales o cuadráticos. Aunque el cálculo eventualmente terminará si el proceso usual de generación de s-polinomios seguido de la reducción de los mismos continúa, a veces es mucho más rápido cancelar la ejecución del algoritmo y reinicializar el cálculo añadiendo los polinomios recién encontrados a los polinomios dados en la entrada. Esto es particularmente cierto cuando la base actual tiene ya un número considerable de polinomios y la aparición de nuevos reductores provocaría que prácticamente toda la base ahora pueda reducirse. Esta misma estrategia es utilizada por Magma, pero desconocemos cuál es la política que utiliza para reinicializar el cálculo. Nosotros reinicializamos el cálculo una única vez cuando aparecen $\lceil \sqrt{n} \rceil$ polinomios lineales, donde n es el número de variables del sistema.

Por otra parte, nuestra implementación puede detectar si la base actual ya es una base de Gröbner al poder decidir si s-polinomios de grado mayor al grado límite de la estrategia grado-truncada son redundantes. El cumplimiendo de los criterios puede determinarse únicamente

examinando los monomios líderes de las parejas polinomiales y, en el caso del criterio de la cadena, las parejas relacionadas deben estar formadas por elementos de la base actual. Gracias a esto y a que nuestra implementación de los criterios de Buchberger no requiere tener explícitamente una lista de parejas pendientes sino que construye un orden entre parejas relacionadas, se pueden reevaluar sin restricciones los criterios de Buchberger sobre todas las parejas polinomiales para determinar si el grado límite en verdad tiene que incrementarse.

3.2. Bases de Gröbner booleanas de sistemas resueltos

Aunque el cálculo de bases de Gröbner se usa casi siempre como método de resolución para sistemas de ecuaciones no lineales, ocasionalmente lo que se desea calcular es la base de Gröbner en sí. Algunos problemas algebraicos tales como decidir si dos ideales son iguales o si un polinomio pertenece a un ideal se pueden resolver usando bases de Gröbner: el primero mediante la comparación de las bases de Gröbner reducidas de los ideales y el segundo mediante la reducción del polinomio sobre la base, la cual es cero si y sólo si el polinomio pertenece al ideal. Desafortunadamente, muchas de las implementaciones mencionadas en el estado de la técnica son incapaces de calcular bases de Gröbner booleanas para algunos sistemas con pocas variables, teniendo problemas incluso a partir de veinte variables.

Durante el desarrollo de esta tesis descubrimos dos maneras de calcular bases de Gröbner sin hacer uso de los fundamentos teóricos del algoritmo de Buchberger. En su lugar, usamos directamente la definición de un ideal y asumimos que conocemos el conjunto completo de soluciones (o no soluciones) del sistema dado. En esta sección describimos los dos algoritmos diseñados. Uno de ellos trabaja a partir del conjunto de soluciones y el otro trabaja a partir del conjunto de no soluciones. Dependiendo del tamaño de los conjuntos correspondientes, uno puede ser más eficiente que el otro. Ya que ambos algoritmos están basados en la misma idea, primero explicamos su fundamento para después describir cada algoritmo individualmente.

3.2.1. Ideas principales de los algoritmos para sistemas resueltos

A cada polinomio booleano p con n variables le corresponde una tabla de verdad, la cual puede construirse evaluándolo con las 2^n combinaciones de valores de verdad para sus variables. Si p pertenece a un ideal $\mathcal{I} \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n)$, por definición se tiene que los productos $p \cdot q$ también pertenecen a $\mathcal{I}$ para toda $q \in \mathbb{B}(x_1, x_2, \ldots, x_n)$. Sea x una asignación de valores de verdad para $x_1, x_2, \ldots, x_n$, la tabla de verdad de $p \cdot q$ puede construirse a partir de las tablas de verdad de p y q donde $(p \cdot q)(x) = p(x) \wedge q(x)$. De manera similar se tiene por definición que $p + p' \in \mathcal{I}$ para todo $p, p' \in \mathcal{I}$ y la tabla de verdad de $p \oplus p'$ se puede construir observando

que $(p \oplus p')(x) = p(x) \oplus p'(x)$ para toda asignación x. Cabe mencionar que la tabla de verdad de $p \cdot q$ tiene un subconjunto de los unos de la tabla de p, mientras que la tabla de $p \cdot p'$ nunca tiene un uno que no aparezca en alguna de las tablas de p y p'.

Sea P un sistema de ecuaciones dado como conjunto de polinomios booleanos $p_1, p_2, \ldots, p_m$ implícitamente igualados a cero, definimos τ_P como la tabla de verdad que tiene ceros en las filas correspondientes a los valores de verdad que evalúan a cero para todo $p \in P$ y que tiene unos en el resto de las filas. Conceptualmente, dicha tabla es la disyunción inclusiva de las tablas de verdad de $p \in P$, pero la definición de un ideal adaptada al álgebra booleana sólo permite aplicar conjunciones y disyunciones exclusivas. Deseamos evidenciar que el polinomio t cuya tabla de verdad es τ_P también pertenece a $\mathcal{I}(P)$. Por una parte, observamos que la disyunción exclusiva entre tablas de verdad que no tienen unos en común es equivalente a su disyunción inclusiva. Por otra parte, cualquier uno de τ_P proviene de algún polinomio $p \in P$ que evalúa a uno bajo la asignación correspondiente de valores de verdad. Dado el uno de la i-ésima fila de τ_P y un polinomio p cuya tabla lo incluye, es posible construir un elemento del ideal cuya tabla tenga sólo ese uno escogiendo un $q \in \mathbb{B}(x_1, x_2, \ldots, x_n)$ adecuado de modo que $p \cdot q$ cumpla la propiedad buscada. Una elección trivial de q es un polinomio cuya tabla tenga exactamente dicho uno y en este caso $p \cdot q = q$. En otras palabras, si τ_P tiene un uno en la i-ésima fila, entonces el polinomio cuya tabla tiene un uno sólo en la i-ésima fila también pertenece a $\mathcal{I}(P)$. Denotando como q_i al polinomio cuya tabla de verdad tiene un uno sólo en la i-ésima fila y como R^1_P al conjunto de las filas de τ_P que tienen unos, se cumple que $t = \sum_{i \in R^1_P} q_i \in \mathcal{I}(P)$. Ya que la tabla de cualquier $p \in \mathcal{I}(P)$ tiene un subconjunto de los unos de τ_P, entonces todo $p \in \mathcal{I}(P)$ es la suma de un subconjunto de $\{q_i : i \in R^1_P\}$.

3.2.2. Construcción a partir del conjunto de soluciones

Cuando un sistema P tiene pocas soluciones, la tabla de verdad τ_P tiene pocas filas con ceros y ningún polinomio de la base de Gröbner de $\mathcal{I}(P)$ puede tener una tabla de verdad con un uno que no esté en τ_P. Verificar que un polinomio q arbitrario pertenece a la base se vuelve trivial, pues q debe evaluar a cero para todas las asignaciones cuyas filas en τ_P tienen cero. Nuestro algoritmo construye directamente la base de Gröbner reducida de P intentando fabricar polinomios de la base cuyos monomios líderes sean irreducibles sobre la base actual. Esto se hace comenzando por los menores monomios líderes del orden monomial elegido.

Nuestro algoritmo mantiene en todo momento el conjunto G de polinomios en la base actual y el conjunto F de polinomios que se fabricaron pero no se pudieron agregar a la base. Cuando se fabrica el polinomio q con monomio líder m, se verifica inicialmente si $q = m$ puede pertenecer a G. En caso afirmativo, q se agrega a G y el proceso se repite para el

siguiente monomio líder en el orden que sea irreducible sobre elementos de G. En otro caso, debemos considerar que la suma $q' = q + \sum_{f \in F'} f$ con $F' \subseteq F$ tal vez pueda agregarse a G si la disyunción exclusiva de las tablas de verdad de los sumandos cumple la propiedad buscada. Dado que los elementos de F fueron generados antes que q, se cumple que $lm(q) > lm(f)$ para todo $f \in F$ y entonces $lm(q') = lm(q)$. Para determinar cuál $F' \subseteq F$ debe sumarse a q, se procede como sigue. Sea $E = R^1_q \cap R^0_P$ el conjunto de las filas con unos que obstruyen la pertenencia de q en G y sea $e_{máx} = \text{máx}\{E\}$. Si existe algún $f \in F$ tal que $e_{máx} = \text{máx}\{R^1_f\}$ entonces $f \in F'$ y $e_{máx}$ se elimina de E, repitiendo el proceso hasta que E esté vacío o no exista la f buscada. Si E es vacío entonces q' se agrega a G y en caso contrario se agrega a F. Observemos que si q' se agrega a F, se sabe que $\text{máx}\{R^1_{q'} \cap R^0_P\} \neq \text{máx}\{R^1_f \cap R^0_P\}$ para todo $f \in F$, por lo que los elementos de F forman un conjunto linealmente independiente con respecto a los unos que obstruyen su pertenencia en G, por lo que $|F| \leq R^0_P$.

Algoritmo 3.7 Construcción a partir de las soluciones del sistema

```
función PorSoluciones(P ⊆ 𝔹(x_1, x_2, ..., x_n))
    G ← ∅, F ← ∅
    para i ← 0 ... 2^n − 1
        q ← monomio(i)
        si ∄g ∈ G : g|q entonces
            E ← R^1_q ∩ R^0_P
            mientras E ≠ ∅ ∧ ∃f ∈ F : e = máx(R^1_f)
                e ← máx{E}
                q ← q + (f ∈ F : e = máx(R^1_f))
                E ← E \ e
            si E = ∅ entonces
                G ← G ∪ q
            si no
                F ← F ∪ q
    regresa G
```

La base de Gröbner construida está reducida porque todos los monomios que eran reducibles sobre elementos de la base fueron ignorados. Por lo tanto, éstos no aparecen en polinomios de G ni en polinomios de F. Este algoritmo toma tiempo $\mathcal{O}(2^n + |G \cup F||F|^2)$ ya que construye la tabla de verdad de P y verifica la independencia lineal de las obstrucciones por unos de $|G \cup F|$ polinomios sobre $|F|$ elementos de a lo mucho $|F|$ monomios cada uno.

Posterior al diseño de este algoritmo, notamos que el algoritmo FGLM, el cual transforma una base de Gröbner calculada bajo un orden monomial a una bajo un orden distinto, está basado en la misma idea. La diferencia entre el algoritmo FGLM y nuestro algoritmo es que

FGLM usa la base de Gröbner en el orden monomial original para verificar si un polinomio, fabricado según el nuevo orden pertenece o no al ideal [21]. Nuestro algoritmo puede verse como una especialización booleana de FGLM que no cambia el orden de una base de Gröbner preexistente sino que la emite directamente a partir de la tabla de verdad del sistema.

3.2.3. Construcción a partir del conjunto de no soluciones

Calcular la base de Gröbner reducida de un conjunto de polinomios es deseable ya que es única y es la más compacta. Sin embargo, basta con que una base tenga un reductor para cualquier polinomio del ideal para que la base sea de Gröbner. Sea P el sistema de polinomios booleanos y $M = \{lm(p) : p \in \mathcal{I}(P)\}$. Una manera de calcular la base de Gröbner G del sistema es exponer un polinomio del ideal para cada $m \in M$ distinto, de modo que cualquier $p \in \mathcal{I}(P)$ pueda reducirse sobre el $g \in G$ con el mismo monomio líder mediante la suma $p' = p + g$. Sea q_i el polinomio cuya tabla de verdad tiene un uno sólo en la i-ésima fila, sabemos que todo $p \in \mathcal{I}(P)$ es una combinación lineal 0-1 de elementos $Q = \{q_i : i \in R^1_P\}$ y por lo tanto Q es algún tipo de base de $\mathcal{I}(P)$, pero no necesariamente es una base de Gröbner. Si G es una base de Gröbner y además todo $p \in \mathcal{I}(P)$ es una combinación lineal 0-1 de elementos de G, entonces $|Q| = |G| = |M|$. Si construimos una matriz para Q tal que las filas denoten los polinomios del conjunto, las columnas denoten los monomios booleanos listados de mayor a menor según el orden monomial y las celdas almacenen los coeficientes de los monomios en los polinomios de Q, podemos escalonar la matriz para que los polinomios del resultado tengan monomios líderes distintos y formen una base de Gröbner.

La construcción y el escalonamiento de la matriz se pueden realizar de manera eficiente al darnos cuenta de que cada q_i tiene una estructura peculiar. Dada una tabla de verdad, cada fila corresponde con una fórmula booleana de la forma $(w_1 \oplus x_1) \wedge (w_2 \oplus x_2) \wedge \cdots \wedge (w_n \oplus x_n)$ donde diremos que x_i está afirmada si $w_i = 0$. Nombraremos como la i-ésima fila a la fila que tenga la i-ésima permutación lexicográfica de $(w_1, w_2, \ldots, w_n)$. Para obtener el polinomio q_i, debemos convertir la fórmula de su fila a la forma normal algebraica. Sea A_i el conjunto de las variables afirmadas en la i-ésima fila y m_X el monomio resultante de multiplicar los elementos del conjunto de variables X, se tiene que $q_i = \sum m_X : A_i \subseteq X \subseteq \{x_1, x_2, \ldots, x_n\}$.

Observemos que el monomio $x_1 x_2 \ldots x_n \in q_i$ para todo $q_i \in Q$, que $m_{A_i} \in q_i$ y que dichos monomios son el monomio líder y el monomio mínimo de q_i respectivamente, lo cual es independiente del orden monomial elegido. Nuestra implementación no construye la matriz de Q de golpe sino que agrega los q_i por lotes, comenzando por aquéllos con monomios mínimos más grandes y realizando el escalonamiento de la matriz cada vez que se agrega un nuevo lote. Con esta estrategia, la distancia entre el monomio líder y el monomio mínimo

de las primeras filas de la matriz es pequeña, lo cual se traduce en filas pivote con pocos elementos. Las filas agregadas posteriormente tienen una distancia cada vez mayor entre sus monomios líder y mínimo, pero los pivotes previos eliminan los monomios más grandes sin introducir monomios menores. Nosotros evitamos modificar filas que ya son pivotes y entonces la matriz queda en una forma escalonada no necesariamente reducida. La ventaja es que los pivotes pueden almacenarse en un formato comprimido de sólo lectura y nosotros usamos los árboles de prefijos que implementamos para la variante del algoritmo F4.

Algoritmo 3.8 Construcción a partir de las no soluciones del sistema

función PorNoSoluciones($P \subseteq \mathbb{B}(x_1, x_2, \ldots, x_n), k \in \mathbb{N}$)
  $G \leftarrow \emptyset$
  $v_f \leftarrow \sum_{i=0}^{d} \binom{n}{i}$
  mientras $v_f \neq 0$
    $v_i \leftarrow v_f - \text{mín}(v_f, k)$
    para cada $i \in R_P^1 \cap [v_i, v_f)$
      $q_i \leftarrow \sum m_X : A_i \subseteq X \subseteq \{x_1, x_2, \ldots, x_n\}$
      $G \leftarrow G \cup q_i$
    $G \leftarrow$ *escalonamiento*(G)
    $v_f \leftarrow v_i$
  regresa G

Para calcular la base de Gröbner reducida, basta con calcular la forma que tendrían las filas correspondientes a polinomios irreducibles si la matriz se hubiera llevado a su forma escalonada reducida. Nuestra implementación hace esto sobre la base de Gröbner no reducida. El algoritmo completo toma tiempo $\mathcal{O}(2^n + |Q|2^{2n})$ ya que construye la tabla de verdad de P y realiza el escalonamiento de una matriz con $|Q| = R_P^1$ polinomios de $\mathcal{O}(2^n)$ monomios cada uno, por lo que es eficiente cuando el número de no soluciones R_P^1 es pequeño. Nuestra implementación concurrente del escalonamiento de la matriz es similar a la usada en el proceso de reducción de nuestra variante de F4: a cada núcleo se le asigna un elemento de un conjunto de polinomios, el cual se reduce repetidamente sobre los pivotes existentes hasta que el polinomio reducido se vuelve un pivote nuevo. Ya que el conjunto de polinomios es linealmente independiente, ningún polinomio reducirá a cero usando sólo sumas de filas.

Capítulo 4

Instancias y pruebas experimentales

En este capítulo presentamos los resultados experimentales obtenidos al ejecutar nuestros algoritmos y las implementaciones del estado de la técnica sobre un conjunto de instancias. Las instancias que usamos ya están expresadas como sistemas de polinomios booleanos, por lo que primero mostramos un ejemplo sobre cómo resolver un problema de optimización usando bases de Gröbner booleanas. La presentación de los resultados la dividimos en dos secciones: el cálculo de bases de Gröbner para instancias generales mediante algoritmos basados en la teoría de Buchberger y el cálculo de bases de Gröbner para instancias con pocas variables, en la que mostramos la eficiencia de nuestros algoritmos sobre sistemas resueltos. Nuestras implementaciones están desarrolladas en C++. El código se encuentra en el apéndice.

4.1. Ejemplo de optimización con bases de Gröbner

Tal como se explicó en la introducción, un programa lineal entero puede transformarse en un sistema de ecuaciones binarias en tiempo polinomial para resolverse mediante el cálculo de su base de Gröbner booleana. A continuación presentamos una instancia del problema de acoplamiento máximo en una gráfica, el cual consiste en calcular un conjunto con la mayor cantidad de aristas posible tal que en cada vértice no incida más de una arista. Ver figura 4.1.

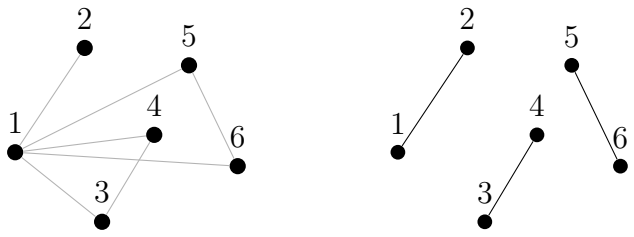


Figura 4.1: Ejemplo de gráfica junto con su acoplamiento máximo.

Este problema se puede modelar fácilmente como un programa entero binario. A cada arista le corresponde una variable binaria que vale 1 si la arista pertenece al acoplamiento y vale 0 en otro caso. El objetivo es maximizar la suma de las variables bajo la restricción de que ningún vértice puede tener dos aristas que pertenezcan al acoplamiento.

$$\begin{aligned}
\text{maximizar} \quad & e_{1,2} + e_{1,3} + e_{1,4} + e_{1,5} + e_{1,6} + e_{3,4} + e_{5,6} \\
\text{sujeto a} \quad & e_{1,2} + e_{1,3} + e_{1,4} + e_{1,5} + e_{1,6} \leq 1 \\
& e_{1,2} \leq 1 \\
& e_{1,3} + e_{3,4} \leq 1 \\
& e_{1,4} + e_{3,4} \leq 1 \\
& e_{1,5} + e_{5,6} \leq 1 \\
& e_{1,6} + e_{5,6} \leq 1 \\
& e_{i,j} \in \mathbb{Z}_2 \text{ para toda arista } (i,j). \text{ donde } i < j
\end{aligned}$$

Transformaremos el modelo anterior a un sistema de ecuaciones binarias no lineales aplicando primero una estrategia ad-hoc sobre las restricciones, para después aplicar un algoritmo general sobre la función objetivo que genera variables auxiliares. Con respecto a las restricciones, observemos que para cualquier pareja de aristas que incidan en el mismo vértice, alguna de las dos debe faltar en el acoplamiento, es decir, el producto de sus variables debe ser cero.

$$\begin{aligned}
\text{maximizar} \quad & e_{1,2} + e_{1,3} + e_{1,4} + e_{1,5} + e_{1,6} + e_{3,4} + e_{5,6} \\
\text{sujeto a} \quad & e_{1,2} \cdot e_{1,3} = 0 \\
& e_{1,2} \cdot e_{1,4} = 0 \\
& e_{1,2} \cdot e_{1,5} = 0 \\
& e_{1,2} \cdot e_{1,6} = 0 \\
& e_{1,3} \cdot e_{1,4} = 0 \\
& e_{1,3} \cdot e_{1,5} = 0 \\
& e_{1,3} \cdot e_{1,6} = 0 \\
& e_{1,4} \cdot e_{1,5} = 0 \\
& e_{1,4} \cdot e_{1,6} = 0 \\
& e_{1,5} \cdot e_{1,6} = 0 \\
& e_{1,3} \cdot e_{3,4} = 0 \\
& e_{1,4} \cdot e_{3,4} = 0 \\
& e_{1,5} \cdot e_{5,6} = 0 \\
& e_{1,6} \cdot e_{5,6} = 0 \\
& e_{i,j} \in \mathbb{Z}_2 \text{ para toda arista } (i,j). \text{ donde } i < j
\end{aligned}$$

Para expresar la función objetivo, conviene observar lo siguiente. Sea $b_1 + b_2$ la suma de dos variables binarias, los dígitos en base binaria del resultado pueden calcularse con las fórmulas en forma normal algebraica $(r_1, r_2) = (b_1 \wedge b_2, b_1 \oplus b_2)$. Esto puede generalizarse cuando alguno de los operandos requiere más de un bit para representarse. Dado que la suma tiene asociatividad izquierda, reescribiremos la expresión

$$((((((e_{1,2} + e_{1,3}) + e_{1,4}) + e_{1,5}) + e_{1,6}) + e_{3,4}) + e_{5,6})$$

usando variables adicionales para representar las sumas intermedias en base binaria. Cada suma intermedia se puede representar con $\lceil \log_2(m) \rceil$ bits, donde m es la magnitud máxima de la suma, y se puede utilizar una variable para cada bit de acarreo, lo que permite que las longitudes de las fórmulas estén acotadas por una constante. En general, el número de variables binarias usadas para expresar la suma de k de estas variables es $\mathcal{O}(k \log_2(k))$. Este algoritmo es similar al descrito en [46]. A continuación presentamos un modelo de ecuaciones binarias donde la función objetivo original está dada por $z = 4t_{6,2} + 2t_{6,1} + 1t_{6,0}$. La base de Gröbner booleana se debe calcular con un orden monomial tal que las variables libres puedan decidirse comenzando por aquéllas que aparecen en la función objetivo y entre éstas, primero por las que correspondan con los bits más altos.

$$
\begin{aligned}
\text{maximizar} \quad & (t_{6,2}, t_{6,1}, t_{6,0}) \\
\text{sujeto a} \quad & e_{1,2} \cdot e_{1,3} = 0 \quad e_{1,2} \cdot e_{1,4} = 0 \quad e_{1,2} \cdot e_{1,5} = 0 \quad e_{1,2} \cdot e_{1,6} = 0 \\
& e_{1,3} \cdot e_{1,4} = 0 \quad e_{1,3} \cdot e_{1,5} = 0 \quad e_{1,3} \cdot e_{1,6} = 0 \\
& e_{1,4} \cdot e_{1,5} = 0 \quad e_{1,4} \cdot e_{1,6} = 0 \quad e_{1,5} \cdot e_{1,6} = 0 \\
& e_{1,3} \cdot e_{3,4} = 0 \quad e_{1,4} \cdot e_{3,4} = 0 \quad e_{1,5} \cdot e_{5,6} = 0 \quad e_{1,6} \cdot e_{5,6} = 0 \\
& s_{1,0} = e_{1,2} \quad s_{1,1} = 0 \quad s_{1,2} = 0 \quad s_{2,0} = e_{1,3} \quad s_{2,1} = 0 \quad s_{2,2} = 0 \\
& s_{3,0} = e_{1,4} \quad s_{3,1} = 0 \quad s_{3,2} = 0 \quad s_{4,0} = e_{1,5} \quad s_{4,1} = 0 \quad s_{4,2} = 0 \\
& s_{5,0} = e_{1,6} \quad s_{5,1} = 0 \quad s_{5,2} = 0 \quad s_{6,0} = e_{3,4} \quad s_{6,1} = 0 \quad s_{6,2} = 0 \\
& s_{7,0} = e_{5,6} \quad s_{7,1} = 0 \quad s_{7,2} = 0 \\
& t_{1,0} = s_{1,0} \oplus s_{2,0} \qquad c_{1,0} = s_{1,0} \cdot s_{2,0} \\
& t_{1,1} = s_{1,1} \oplus s_{2,1} \oplus c_{1,0} \qquad c_{1,1} = s_{1,1} \cdot s_{2,1} \oplus s_{1,1} \cdot c_{1,0} \oplus s_{2,1} \cdot c_{1,0} \\
& t_{1,2} = s_{1,2} \oplus s_{2,2} \oplus c_{1,1} \\
& t_{2,0} = s_{3,0} \oplus t_{1,0} \qquad c_{2,0} = s_{3,0} \cdot t_{1,0} \\
& t_{2,1} = s_{3,1} \oplus t_{1,1} \oplus c_{2,0} \qquad c_{2,1} = s_{3,1} \cdot t_{1,1} \oplus s_{3,1} \cdot c_{2,0} \oplus t_{1,1} \cdot c_{2,0}
\end{aligned}
$$

$$
\begin{aligned}
&t_{2,2} = s_{3,2} \oplus t_{1,2} \oplus c_{2,1} \\
&t_{3,0} = s_{4,0} \oplus t_{2,0} && c_{3,0} = s_{4,0} \cdot t_{2,0} \\
&t_{3,1} = s_{4,1} \oplus t_{2,1} \oplus c_{3,0} && c_{3,1} = s_{4,1} \cdot t_{2,1} \oplus s_{4,1} \cdot c_{3,0} \oplus t_{2,1} \cdot c_{3,0} \\
&t_{3,2} = s_{4,2} \oplus t_{2,2} \oplus c_{3,1} \\
&t_{4,0} = s_{5,0} \oplus t_{3,0} && c_{4,0} = s_{5,0} \cdot t_{3,0} \\
&t_{4,1} = s_{5,1} \oplus t_{3,1} \oplus c_{4,0} && c_{4,1} = s_{5,1} \cdot t_{3,1} \oplus s_{5,1} \cdot c_{4,0} \oplus t_{3,1} \cdot c_{4,0} \\
&t_{4,2} = s_{5,2} \oplus t_{3,2} \oplus c_{4,1} \\
&t_{5,0} = s_{6,0} \oplus t_{4,0} && c_{5,0} = s_{6,0} \cdot t_{4,0} \\
&t_{5,1} = s_{6,1} \oplus t_{4,1} \oplus c_{5,0} && c_{5,1} = s_{6,1} \cdot t_{4,1} \oplus s_{6,1} \cdot c_{5,0} \oplus t_{4,1} \cdot c_{5,0} \\
&t_{5,2} = s_{6,2} \oplus t_{4,2} \oplus c_{5,1} \\
&t_{6,0} = s_{7,0} \oplus t_{5,0} && c_{6,0} = s_{7,0} \cdot t_{5,0} \\
&t_{6,1} = s_{7,1} \oplus t_{5,1} \oplus c_{6,0} && c_{6,1} = s_{7,1} \cdot t_{5,1} \oplus s_{7,1} \cdot c_{6,0} \oplus t_{5,1} \cdot c_{6,0} \\
&t_{6,2} = s_{7,2} \oplus t_{5,2} \oplus c_{6,1}
\end{aligned}
$$

A continuación se muestra una base de Gröbner reducida, calculada bajo un orden monomial que permite realizar la optimización. Los polinomios se listan bajo el mismo orden.

$$
\begin{aligned}
&\{c_{1,0}, c_{1,1}, c_{2,0}, c_{2,1}, c_{3,0}, c_{3,1}, c_{4,0}, c_{4,1}, c_{5,0} + t_{5,1}, c_{5,1}, c_{6,0} + t_{5,1} + t_{6,1}, c_{6,1}, e_{1,2} + s_{2,0} + \\
&t_{1,0}, e_{1,3} + s_{2,0}, e_{1,4} + t_{1,0} + t_{2,0}, e_{1,5} + t_{2,0} + t_{3,0}, e_{1,6} + t_{3,0} + t_{4,0}, e_{3,4} + t_{4,0} + t_{5,0}, e_{5,6} + \\
&t_{5,0} + t_{6,0}, s_{1,0} + s_{2,0} + t_{1,0}, s_{1,1}, s_{1,2}, s_{2,0}t_{1,0} + s_{2,0}, s_{2,0}t_{2,0} + s_{2,0}, s_{2,0}t_{3,0} + s_{2,0}, s_{2,0}t_{4,0} + \\
&s_{2,0}, s_{2,0}t_{5,0} + s_{2,0}, s_{2,0}t_{5,1}, s_{2,0}t_{6,0} + s_{2,0}t_{6,1} + s_{2,0}, s_{2,1}, s_{2,2}, s_{3,0} + t_{1,0} + t_{2,0}, s_{3,1}, s_{3,2}, s_{4,0} + \\
&t_{2,0} + t_{3,0}, s_{4,1}, s_{4,2}, s_{5,0} + t_{3,0} + t_{4,0}, s_{5,1}, s_{5,2}, s_{6,0} + t_{4,0} + t_{5,0}, s_{6,1}, s_{6,2}, s_{7,0} + t_{5,0} + \\
&t_{6,0}, s_{7,1}, s_{7,2}, t_{1,0}t_{2,0} + t_{1,0}, t_{1,0}t_{3,0} + t_{1,0}, t_{1,0}t_{4,0} + t_{1,0}, t_{1,0}t_{5,0} + t_{1,0} + t_{2,0}t_{6,1} + t_{4,0}t_{6,1} + \\
&t_{5,1}, t_{1,0}t_{5,1} + t_{2,0}t_{6,1} + t_{4,0}t_{6,1} + t_{5,1}, t_{1,0}t_{6,0} + t_{1,0}t_{6,1} + t_{1,0} + t_{6,0}t_{6,1}, t_{1,1}, t_{1,2}, t_{2,0}t_{3,0} + \\
&t_{2,0}, t_{2,0}t_{4,0} + t_{2,0}, t_{2,0}t_{5,0} + t_{2,0}t_{6,1} + t_{2,0} + t_{4,0}t_{6,1} + t_{5,1}, t_{2,0}t_{5,1} + t_{2,0}t_{6,1} + t_{4,0}t_{6,1} + \\
&t_{5,1}, t_{2,0}t_{6,0} + t_{2,0}t_{6,1} + t_{2,0} + t_{6,0}t_{6,1}, t_{2,1}, t_{2,2}, t_{3,0}t_{4,0} + t_{3,0}, t_{3,0}t_{5,0} + t_{3,0}t_{6,1} + t_{3,0} + t_{4,0}t_{6,1} + \\
&t_{5,1}, t_{3,0}t_{5,1} + t_{3,0}t_{6,1} + t_{4,0}t_{6,1} + t_{5,1}, t_{3,0}t_{6,0} + t_{3,0}t_{6,1} + t_{3,0} + t_{6,0}t_{6,1}, t_{3,1}, t_{3,2}, t_{4,0}t_{5,0} + \\
&t_{4,0} + t_{5,1}, t_{4,0}t_{5,1} + t_{5,1}, t_{4,0}t_{6,0} + t_{4,0}t_{6,1} + t_{4,0} + t_{6,0}t_{6,1}, t_{4,1}, t_{4,2}, t_{5,0}t_{5,1}, t_{5,0}t_{6,0} + t_{5,0} + \\
&t_{5,1} + t_{6,1}, t_{5,0}t_{6,1} + t_{5,1} + t_{6,1}, t_{5,1}t_{6,0} + t_{6,0}t_{6,1}, t_{5,1}t_{6,1} + t_{5,1}, t_{5,2}, t_{6,2}\}
\end{aligned}
$$

Todos los polinomios están implícitamente igualados a cero, por lo que $t_{6,2} = 0$ y $t_{5,2} = 0$ por las últimas dos ecuaciones de la base. La antepenúltima ecuación permite elegir $t_{6,1} = 1$

mientras que $t_{5,1}$ sigue libre. La ecuación previa permite elegir $t_{6,0} = 1$ y esto fuerza el resto de las variables. Al final se tiene que $e_{1,2} = e_{3,4} = e_{5,6} = 1$ y $e_{1,3} = e_{1,4} = e_{1,5} = e_{1,6} = 0$.

4.2. Instancias de prueba

Nuestro conjunto de instancias de prueba incluyen el Reto HFE 1, así como instancias con pocas variables provenientes del sistema criptográfico Aceite-Vinagre No-Equilibrado [33]. Aunque consideramos que el Reto HFE 1 es la instancia más difícil de resolver en términos generales (independientemente de si la técnica de resolución es el cálculo de bases de Gröbner), las instancias PK del sistema criptográfico Aceite-Vinagre No-Equilibrado se caracterizan por tener bases de Gröbner muy grandes y con polinomios de grado alto, lo cual dificulta mucho el cálculo de las bases. También generamos cinco ideales booleanos mediante la negación de la tabla de verdad de cada instancia PK. Dado que no contamos con un conjunto de polinomios de entrada para estos ideales pero sí con su tabla de verdad, éstos se usarán en las pruebas de rendimiento de los algoritmos para sistemas resueltos. Ver cuadro 4.1.

Instancia	Variables	Polinomios en la entrada	Polinomios en la salida	Grado de la salida	Soluciones
PK15	15	5	639	5	1008
PK16	16	5	1891	5	2040
PK17	17	5	2069	6	4160
PK18	18	6	3328	5	4160
PK19	19	6	4800	6	8048
PK20	20	6	12493	6	16736
HFE1	80	80	80	2	4
PK15n	15	-	507	11	31760
PK16n	16	-	955	12	63496
PK17n	17	-	1623	12	126912
PK18n	18	-	1581	14	257984
PK19n	19	-	4049	14	516240
PK20n	20	-	6145	15	1031840

Cuadro 4.1: Estadísticas generales de las instancias de prueba.

Por otra parte, las instancias descritas como conjuntos de polinomios se resolvieron con nuestra variante del algoritmo F4 y presentamos en el cuadro 4.2 algunas estadísticas que son de interés en el caso de algoritmos basados en la teoría de Buchberger y en el algoritmo F4. En todos los casos se utilizó el orden graduado lexicográfico reverso. Los tamaños reportados para las matrices son teóricos e incluyen elementos de la base, filas reductoras y polinomios

por reducir. En la práctica, nuestra variante sólo incluye un subconjunto de las filas reductoras a la vez. En las instancias PK, el grado alcanzado por la estrategia grado-truncada es muy alto y la estrategia es poco efectiva. Adicionalmente, la cantidad de s-polinomios examinados y reducidos es considerable y prácticamente todas las reducciones son reducciones a cero.

Instancia	Grado alcanzado	Evaluaciones de cadena	S-polinomios reducidos	Reducciones a cero	Matriz más grande
PK15	6	100956	18190	16177	19733×9949
PK16	7	527456	44144	40862	54181×26333
PK17	7	530754	77704	72212	68394×41226
PK18	7	1372561	135888	127162	117260×63044
PK19	7	1252466	163824	148538	189816×94184
PK20	8	6983202	377510	351375	456686×263950
HFE1	4	64990	43662	8845	320040×1666981

Cuadro 4.2: Estadísticas de las instancias al resolverlas con la variante del algoritmo F4.

4.3. Cómputo de bases en sistemas no resueltos

En esta sección mostramos el rendimiento de nuestra variante del algoritmo F4. Primero mostramos el comportamiento experimental de nuestra implementación de los criterios de Buchberger, así como del proceso de reducción polinomial en función del número de hilos de ejecución concurrentes y de la saturación de la caché L3 en el caso de la segunda. También mostramos qué tan efectivo es nuestro esquema de compresión de polinomios en comparación con haberlos representado directamente como arreglos de bits. Posteriormente comparamos el uso de recursos de nuestra implementación con las implementaciones del estado de la técnica.

4.3.1. Comportamiento experimental de la variante de F4

En comparación con la verificación secuencial de los criterios de Buchberger descrita por Gebauer y Möller [28], nuestra implementación examina todas las parejas polinomiales concurrentemente y alcanza una aceleración lineal con respecto al número de núcleos físicos utilizados. También presentamos la aceleración alcanzada en un procesador que cuenta con la tecnología *hyper-threading*, la cual simula dos núcleos lógicos por cada núcleo físico. Aunque prácticamente todos los recursos de un núcleo físico se comparten entre sus núcleos lógicos, la unidad aritmético-lógica de cada núcleo físico puede ejecutar múltiples instrucciones en paralelo. Si un hilo lógico no logra hacer uso de todos los recursos disponibles en el núcleo físico, el otro hilo puede aprovechar los recursos restantes. Ver figura 4.2.

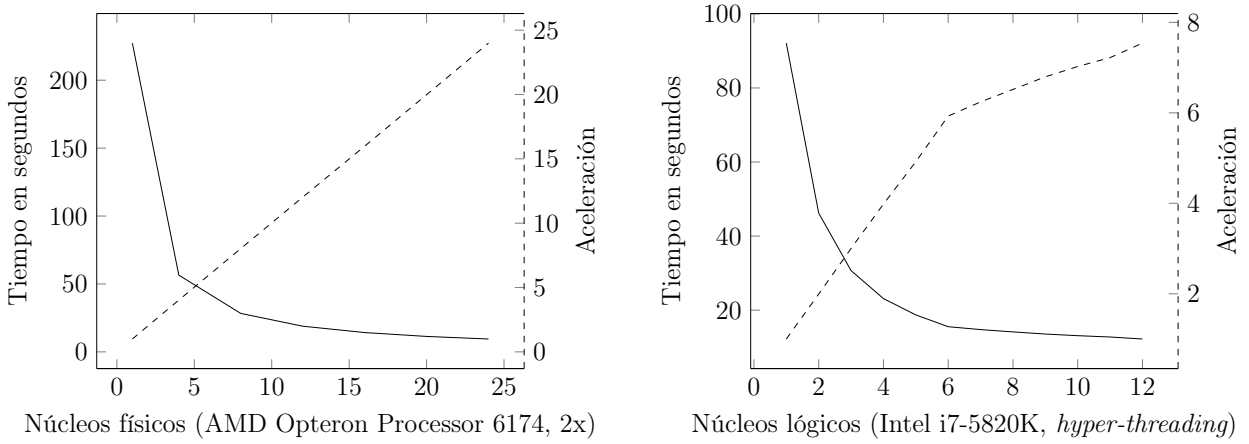


Figura 4.2: Tiempo total usado en la detección de redundancia de s-polinomios para PK20.

Se puede observar que nuestra implementación alcanza una aceleración lineal con respecto al número de núcleos físicos en ambos procesadores. En el caso del procesador Intel i7-5820K, cada núcleo físico cuenta con recursos que un único hilo lógico no alcanza a saturar, pero no con recursos suficientes como para que dos hilos lógicos puedan completar sus cálculos sin intereferir uno con el otro. A pesar de lo anterior, la tecnología *hyper-threading* incrementa de manera perceptible la aceleración en aproximadamente un 30 %.

Por otra parte, el proceso de reducción polinomial contribuye con más del 90 % del tiempo total de ejecución de nuestro algoritmo y su cuello de botella es el tiempo de acceso a memoria. Como se explicó anteriormente, consideramos que es buena idea que las filas reductoras y los polinomios que están siendo reducidos concurrentemente (uno por núcleo lógico) quepan todos de manera justa en la caché L3. Para validar esta decisión, probamos nuestra implementación con la instancia del Reto HFE 1, variando el tamaño del conjunto de reductores activos y utilizando todos los núcleos lógicos disponibles. La cola de polinomios se dividió en grupos de a lo mucho 2048 elementos y cada grupo se redujo en una sola pasada. Las pruebas se realizaron en un procesador Intel i7-5820K corriendo a 3.3GHz, el cual cuenta con 15MB de caché L3 y seis núcleos físicos con la tecnología *hyper-threading* habilitada. Ver figura 4.3.

Cada polinomio de grado 4 que deba reducirse para resolver el Reto HFE 1 ocupa 208372 kilobytes bajo la representación de arreglo de bits, con a lo mucho 1666981 monomios distintos. Aproximadamente 60 de estos arreglos (un polinomio en cada uno de los 12 núcleos lógicos más 48 reductores activos) ocupan 13MB de la caché L3. El resto de la caché L3 se utiliza para la pila de cada hilo en ejecución y para otros costos triviales pero difíciles de evitar, como la memoria del propio código ejecutable. Como ya se había explicado, reducir el número de reductores activos incrementa el número de veces que un polinomio a reducir se lee de memoria principal, ya que éste sólo podrá reducirse pocas veces mientras está en la

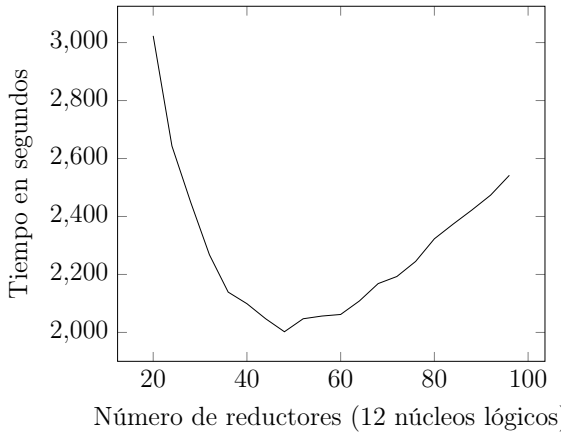


Figura 4.3: Tiempos de ejecución para el Reto HFE 1 con un número variable de reductores.

caché. Por otra parte, aumentar el número de reductores activos causará que los reductores se expulsen mutuamente de la caché L3 al no caber todos, y los reductores expulsados tendrán que ser leídos de memoria principal en su próximo uso. Disminuir el tamaño de los grupos que se reducen una sola pasada para que también quepan en la caché L3 tampoco es buena idea, pues el número de pasadas aumentaría dramáticamente y cada pasada requiere leer de memoria principal los polinomios que generarán las filas reductoras.

También probamos nuestra implementación usando la cantidad óptima de reductores activos pero variando la cantidad de núcleos lógicos utilizados del procesador Intel i7-5820K. Los resultados se muestran en la figura 4.4. En este caso, el uso de la tecnología *hyper-threading* no aporta ningún beneficio y esto se puede explicar como sigue. Nuestra implementación utiliza el conjunto de instrucciones AVX2 que opera sobre registros vectoriales de 256 bits y la mayor parte del trabajo consiste en sumar filas muy largas, por lo que un hilo de ejecución podría solicitar con anticipación la lectura de los subarreglos de 256 bits que debe sumar al no haber dependencias secuenciales. Desafortunadamente, el número de peticiones simultáneas a memoria está limitado por los recursos del núcleo físico. Por ejemplo, cada vez que un dato no se encuentra en la caché L1, el núcleo físico debe apartar un búfer de llenado para traer el dato desde niveles superiores de memoria. Cada núcleo físico del procesador Intel i7-5820K tiene sólo diez búferes de llenado, por lo que los núcleos lógicos estarán compitiendo por recursos correspondientes a peticiones a memoria que un solo hilo puede saturar. Más aún, la caché L3 es compartida por todos los núcleos físicos y coordinar los accesos a la misma impone cierta sobrecarga. Algo similar ocurre con la memoria principal, la cual tiene una tasa de transferencia de datos limitada. En general, las limitaciones con respecto a los accesos a memoria simultáneos provocan que el cálculo no logre acelerarse a un sexto del tiempo de un solo núcleo, a pesar de contar con seis núcleos físicos.

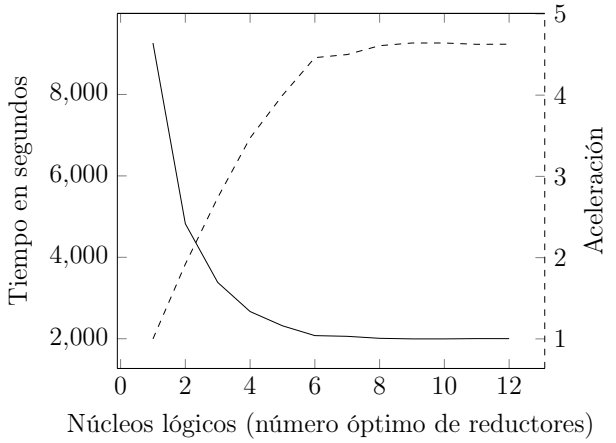
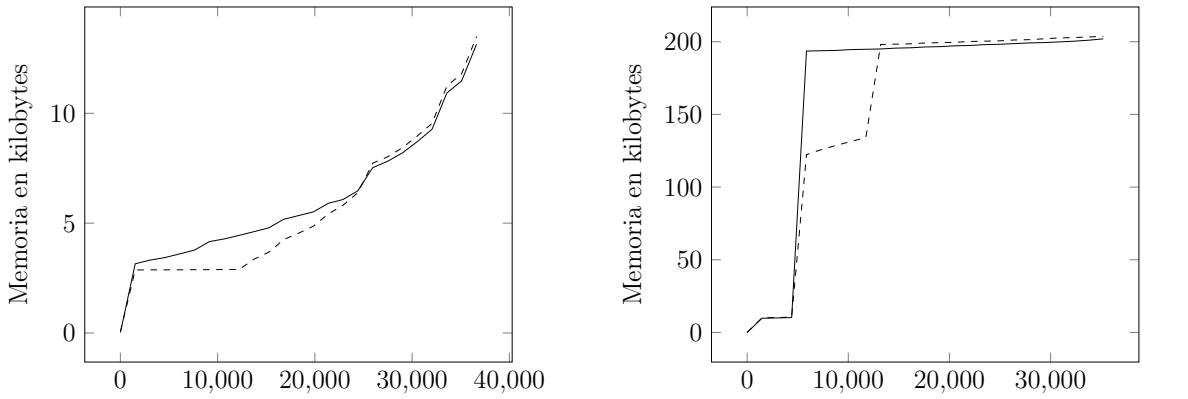


Figura 4.4: Tiempos de ejecución para el Reto HFE 1 con un número variable de núcleos.



(PK20) Uso de memoria de 36,637 polinomios

(HFE1) Uso de memoria de 35,209 polinomios

Figura 4.5: Consumo de memoria de los polinomios que entran a la base durante el cálculo. Se grafica el consumo de cada polinomio (con compresión en punteado) de menor a mayor.

Con respecto al consumo de memoria de los polinomios irreducibles de la base, los cuales no serán modificados durante las reducciones polinomiales, encontramos que nuestro esquema de compresión fue medianamente efectivo. En la figura 4.5 mostramos el consumo de memoria de los polinomios que en algún momento entraron a la base (y que se comprimieron al hacerlo) durante el cálculo de la base de Gröbner para la instancia PK20 y para el Reto HFE 1. Dado que nuestro esquema de compresión sólo almacena las posiciones de los monomios que sí aparecen en el polinomio dado, éste es efectivo si el grado del polinomio es alto pero tiene pocos monomios. Desafortunadamente, la mayoría de los polinomios en ambas instancias de prueba son relativamente densos y tienen casi la mitad de los monomios posibles para su grado. Desde el punto de vista del árbol de prefijos, casi todos los prefijos de 3 bytes aparecen en el árbol y casi todos los nodos hoja se representan como arreglos de 256 bits. A pesar de

lo anterior, el consumo de memoria de los árboles se comporta adecuadamente dado que los prefijos de las posiciones en el orden monomial se comparten. Como se verá posteriormente, nuestra representación comprimida se compara favorablemente con otras representaciones que se especializan en el caso de polinomios densos o en el caso de polinomios dispersos.

4.3.2. Comparación de rendimiento con otras implementaciones

Comparamos el tiempo de ejecución y el consumo de memoria de nuestra implementación con PolyBori, FGb y Magma. Se usaron las instancias PK15, PK16, PK17, PK18, PK19, PK20 y el Reto HFE 1. Aunque consideramos que BooleanGB ha sido una importante fuente de inspiración para la realización de este trabajo, la implementación disponible como un módulo de Macaulay2 1.9.2 es demasiado lenta como para incluirla en las pruebas, requiriendo más de una hora de cómputo incluso para el caso de prueba más sencillo. Aunque usar Macaulay2 directamente sin BooleanGB baja el tiempo de ejecución a la mitad, el consumo de memoria empeora terriblemente, requiriendo varios gigabytes incluso para PK15. En contraste, la peor implementación que sí comparamos requiere un minuto de cálculo y 350MB de consumo de memoria. Algo similar pasa con una implementación reciente del algoritmo F4 denominada OpenF4, la cual no logra calcular PK15 sin haber consumido al menos 8GB de memoria [12].

Nuestra implementación, FGb 1.68 y PolyBori 0.8.3 se probaron en una computadora con procesador Intel i7-5820K corriendo a 3.3GHz y con 8GB de RAM. Sin embargo, la no disponibilidad de una instalación de Magma nos obliga a depender de reportes externos y realizar pruebas con limitaciones de software y en hardware desconocido. Para el Reto HFE 1 se reportan los resultados de la ejecución de Magma 2.20 corriendo en un procesador Intel Xeon E5-1650 a 3.2GHz y con 64GB de RAM [44]. Las instancias PK se resolvieron en hardware desconocido vía la calculadora gratuita de Magma V2.22-7, la cual impone un límite de 120 segundos y 380MB de consumo de memoria [5]. Ver cuadro 4.3.

Nuestra implementación es muy rápida y consume la menor cantidad de memoria de manera consistente. Nuestro consumo de memoria podría reducirse aún más si partimos el conjunto de polinomios por reducir en grupos más pequeños (por ejemplo, 512 en lugar de 2048), aunque esto incrementaría ligeramente el tiempo de ejecución. La variante dispersa de Magma es bastante más lenta que nuestra implementación, además de que consume una cantidad considerable de memoria. Creemos que la variante dispersa de Magma simplemente lista las posiciones de los monomios que aparecen en el polinomio usando un entero para cada uno, pero los polinomios de estas instancias contienen demasiados monomios. Por otra parte, la variante densa de Magma es por mucho la más rápida en el Reto HFE 1. Esta variante utiliza algoritmos asintóticamente más rápidos para escalonar matrices densas como

Instancia	PolyBori	FGb	Magma (disperso)	Magma (denso)	Variante de F4
PK15	60.7s 350MB	2.4s 108MB	1.5s 64MB	1.3s 32MB	0.8s 15MB
PK16	518.1s 991MB	18.1s 202MB	7.2s 256MB	5.7s 62MB	5.3s 18MB
PK17	1169.4s 2140MB	30.2s 247MB	* >380MB	20.6s 128MB	15.4s 32MB
PK18	2976.5s 4229MB	70.3s 359MB	* >380MB	37.1s 155MB	41.9s 50MB
PK19	5366.5s 6829MB	388.5s 856MB	* >380MB	* >225MB	101.2s 122MB
PK20	* >8192MB	952.2s 1370MB	* >380MB	* >225MB	949.7s 260MB
HFE1	* >8192MB	>9000s >8192MB	3477.1s 14029MB	447.4s 5836MB	2002.4s 5981MB

Cuadro 4.3: Pruebas de rendimiento para sistemas no resueltos (* significa que el tiempo no se reporta porque el programa consume rápidamente toda la memoria disponible).

las que aparecen en Reto HFE 1, pero éstos no se utilizan ni en la variante dispersa ni en nuestra implementación [45]. Magma también reporta que su implementación de F4 se ejecuta usando un único núcleo, pero no sabemos si esto es cierto para todo el programa o sólo para el código correspondiente a la teoría de bases de Gröbner que invoca las rutinas de álgebra lineal. Nosotros creemos que, además de usar algoritmos asintóticamente más rápidos para matrices densas, la implementación densa de Magma utiliza las cachés en todos los niveles (L1, L2 y L3) de manera inteligente. Mejorando un poco nuestra implementación del proceso de reducción polinomial podríamos mejorar considerablemente nuestro tiempo de ejecución total, pero el rendimiento de los algoritmos de álgebra lineal implementados en Magma ha sido un referente durante décadas a pesar de ser de código cerrado.

Con respecto a las instancias PK, si bien nuestra implementación es la más rápida y la que consume la menor cantidad de memoria en prácticamente todas las instancias, tanto la variante densa de Magma como FGb logran completar los cálculos con un uso comparable de recursos y en particular de tiempo. Debemos resaltar que a pesar de que FGb es considerablemente menos eficiente que nuestra implementación y que Magma para el Reto HFE 1, las instancias PK tienen una cantidad muy alta de polinomios redundantes que no logran ser detectados por los criterios de Buchberger y es probable que el algoritmo F5 de FGb sí

los detecte. Dada la dificultad de calcular bases de Gröbner para las instancias PK, creemos que instancias PK con muchas más variables no podrán resolverse usando bases de Gröbner.

Realizamos una prueba adicional para verificar la dificultad de resolver el Reto HFE 1, la cual consistió en intentar resolver la instancia con el solucionador de programación entera mixta Gurobi 7.0 [30]. Se generaron dos versiones de la instancia para resolverse con Gurobi, una con restricciones lineales y la otra con restricciones cuadráticas. En ambas versiones, una fórmula de la forma $t_1 \oplus t_2 \oplus \cdots \oplus t_k = b$ se reescribe como $t_1 + t_2 + \cdots + t_k = 2c + b$ donde $c \in \mathbb{Z}$ y $b \in \mathbb{Z}_2$. Para convertir restricciones cuadráticas en restricciones lineales, el producto $x_1 x_2$ se reescribe como $x_1 + x_2 - 1 \leq x_{1,2} \leq x_1, x_2$ con $x_{1,2} \in \mathbb{Z}_2$. La estrategia por omisión de Gurobi, que es explorar de manera inteligente el espacio de búsqueda, no pudo encontrar ninguna solución con ambas versiones del modelo después de varias horas de cómputo y decenas de gigabytes de consumo de memoria. Cuando habilitamos todas las heurísticas disponibles en Gurobi para la búsqueda de soluciones factibles, Gurobi pudo encontrar una de las cuatro soluciones en menos de un minuto, pero delegó la búsqueda del resto de las soluciones a la exploración del espacio de búsqueda, la cual no tuvo éxito. Gurobi sólo pudo encontrar una de las soluciones factibles cuando se utilizó la versión del modelo con restricciones lineales.

4.4. Cómputo de bases en sistemas resueltos

A continuación mostramos una comparación entre nuestra variante del algoritmo F4 y los algoritmos especializados para calcular bases de Gröbner dado el conjunto de soluciones y no soluciones del sistema. También incluimos el conjunto de instancias PKn, las cuales se construyen a partir de la negación de las tablas de verdad de los sistemas PK. Sólo comparamos los algoritmos especializados con nuestra variante del algoritmo F4, al ser ésta la implementación más rápida que está basada en la teoría de Buchberger. Dado que no podemos construir de manera objetiva un conjunto de polinomios de entrada para las instancias PKn, estas instancias no se resolvieron con nuestra variante del algoritmo F4. Ver cuadro 4.4.

Como se puede observar, el algoritmo que trabaja sobre el conjunto de soluciones construyendo directamente los elementos de la base, es sumamente efectivo para las instancias PK originales, ya que en éstas sólo una pequeña fracción de todas las asignaciones de valores de verdad son soluciones del sistema (un promedio de 2 %). El algoritmo que trabaja sobre el conjunto de no soluciones transformando una base lineal en una base de Gröbner, es bastante más lento en estas instancias pero es comparable en rendimiento con la variante del algoritmo F4. Su principal desventaja es que construye todas las filas de la matriz escalonada para finalmente descartar los polinomios que no pertenecen a la base de Gröbner reducida.

Instancia	Variante de F4	Por soluciones	Por no soluciones
PK15	0.8s 15MB	0.2s 3MB	0.5s 8MB
PK16	5.3s 18MB	1.1s 5MB	3.1s 18MB
PK17	15.4s 32MB	4.3s 8MB	11.6s 41MB
PK18	41.9s 50MB	5.4s 11MB	24.8s 75MB
PK19	101.2s 122MB	23.0s 29MB	179.2s 266MB
PK20	949.7s 260MB	207.7s 116MB	981.546s 876MB
PK15n	- -	325.1s 142MB	0.2s 9MB
PK16n	- -	2009.7s 526MB	0.4s 19MB
PK17n	- -	>3600s >1054MB	2.1s 46MB
PK18n	- -	* *	3.8s 50MB
PK19n	- -	* *	33.5s 348MB
PK20n	- -	* *	234.5s 1103MB

Cuadro 4.4: Pruebas de rendimiento para sistemas resueltos (* significa que no se realizó la prueba por considerar que el programa es demasiado lento).

Con respecto a las instancias PK negadas, el algoritmo que trabaja sobre el conjunto de soluciones es sumamente ineficiente. Dado que este algoritmo construye los elementos de la base secuencialmente al intentar fabricar un polinomio para cada monomio líder y en orden creciente según el orden monomial, si la cantidad de elementos que no están en la base es muy grande entonces el algoritmo simplemente no termina en un tiempo razonable. El algoritmo que trabaja sobre el conjunto de no soluciones es, en las instancias negadas, tan rápido como el mejor algoritmo sobre las instancias originales, pero con un consumo de memoria superior. En este caso, el alto consumo de memoria ocurre porque los polinomios de la base tienen

grado muy alto y también tienen muchos monomios.

También realizamos algunas pruebas calculando bases de Gröbner bajo el orden monomial lexicográfico en lugar del orden graduado lexicográfico reverso. Los resultados fueron mixtos, con algunas instancias calculándose hasta diez veces más rapido pero consumiendo hasta cinco veces más memoria (en particular en las instancias para las que el consumo de memoria ya era alto). Dado que las bases de Gröbner reducidas para las instancias PK tienen elementos de grado alto, creemos que usar el orden graduado lexicográfico reverso no aporta ninguna ventaja y en cambio, el orden lexicográfico genera matrices cuyas entradas tienen menos estructura por columnas pero que se escalonan más rápido, a expensas de tener una mayor cantidad de polinomios irreducibles. Para instancias más difíciles como el Reto HFE 1, para la que el resultado sí se puede calcular con polinomios de grado bajo y usando un orden monomial graduado, no recomendamos utilizar el orden lexicográfico.

Capítulo 5

Conclusiones y trabajo futuro

Las bases de Gröbner son un método de resolución de sistemas de ecuaciones que cada vez es más competitivo. Si bien la naturaleza algebraica del cálculo es a veces difícil de predecir y teóricamente más costoso que el algoritmo de fuerza bruta, conforme las implementaciones del cálculo de bases de Gröbner han mejorado, también han aparecido instancias cuyo mejor método de solución conocido es precisamente el cálculo de dichas bases. En este trabajo presentamos diversos algoritmos e implementaciones de alto rendimiento que, por una parte, han mejorado etapas del cálculo que no habían sido mejorados en décadas y por la otra, compiten seriamente con implementaciones del estado de la técnica de código cerrado. Además, presentamos ideas muy sencillas sobre cómo calcular bases de Gröbner para sistemas con pocas variables que son, por varios órdenes de magnitud, más eficientes tanto en tiempo como en memoria que las implementaciones abiertas más recientes y aplaudidas. Una conclusión a la que podemos llegar de lo anterior, es que el cálculo de las bases de Gröbner aún tiene mucho margen de mejora. Esperamos que el cálculo de bases de Gröbner eventualmente se vuelva una técnica conocida por la gente que resuelve cotidianamente problemas combinatorios.

Curiosamente, una vez que una herramienta se vuelve viable de utilizar, la mayoría de los problemas verdaderamente interesantes surgen y cobra sentido pensar en resolverlos. ¿Qué tanto cambia la base de Gröbner de un sistema al agregar una nueva restricción? ¿Qué tanto se complica el cálculo de una base al agregar el conjunto de polinomios que describen la función objetivo, si de todos modos la base de Gröbner calcula todas las soluciones? ¿Qué operaciones del cálculo de bases de Gröbner podrían acelerarse con hardware especializado? Este fenómeno ha ocurrido de manera repetida a lo largo de la historia con diferentes problemas y tecnologías. Consideramos que el trabajo realizado en esta tesis, en particular nuestra implementación de software libre, abre el camino a esta posibilidad.

Apéndice A

Código fuente de los algoritmos desarrollados

A.1. Algoritmos sobre sistemas resueltos

```
//#define TBB_USE_DEBUG 1
//#define TBB_USE_THREADING_TOOLS 1

#include <tbb/task_scheduler_init.h>
#include <bitset>
#include <fstream>
#include <iostream>
#include <map>
#include <set>
#include <sstream>

#include "lex.h"
#include "grevlex.h"
#include "monomial.h"
#include "polynomial.h"
#include "polynomial_io.h"
#include "sparse_polynomial.h"
#include <base/functional.h>
#include <parallel/algorithm>
```

```
#include <tbb/concurrent_vector.h>
#include <tbb/parallel_for.h>
#include <tbb/parallel_for_each.h>
#include <tbb/parallel_sort.h>
#include <tbb/partitioner.h>
#include <tbb/scalable_allocator.h>
#include <tbb/task.h>
#include <algorithm>
#include <array>
#include <atomic>
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <iostream>
#include <numeric>
#include <shared_mutex>
#include <utility>
#include <tuple>

template<typename P, typename OI, typename C>
inline OI escribe_polinomio(const P& p, OI oi, const C& conv)
{
   p.visit([&](auto i) {
      *oi++ = conv.monomial_of(i);
   });

   return oi;
}

template<std::uint32_t N, typename C>
inline void evalua_polinomio(const std::vector<monomial<N>>& lista
   , std::vector<char>& tabla, const C& conv)
{
   tbb::parallel_for(std::uint32_t(0), base::pow2(N), [&](auto i)
      {
```

```
        auto& res = tabla[i];

        if (res == false) {
            auto verdaderos = conv.monomial_of(i);
            for (const auto& monomio : lista) {
                res ^= greatest_common_divisor(verdaderos, monomio) ==
                    monomio;
            }
        }
    });
}

namespace impl {
    enum status : std::uint8_t { INDEFINIDO = 0, BLOQUEADO = 1,
        DEFINIDO = 2 };

    struct registro_matriz {
        registro_matriz( )
        : estado(INDEFINIDO)
        {
        }

        std::atomic<std::uint8_t> estado;
        sparse_polynomial polinomio;
        std::shared_timed_mutex sincronizacion;
    };
}

template<std::uint32_t N, typename C>
inline std::vector<sparse_polynomial> metodo_matriz(const std::
    vector<char>& tabla, const C& conv)
{
    std::array<tbb::concurrent_vector<std::uint32_t>, N + 1> grupos
        ;
    tbb::parallel_for(std::uint32_t(0), base::pow2(N), [&](auto i)
```

```
        {
        if (tabla[i]) {
            grupos[conv.degree_of(i)].push_back(i);
        }
    });

    std::vector<impl::registro_matriz> torre(base::pow2(N));
    for (std::uint32_t i = N + 1; i > 0; --i) {
std::cerr << "en grado " << i - 1 << " con " << grupos[i - 1].size
    () << "...\n";
        tbb::parallel_for_each(grupos[i - 1], [&](auto raiz) {
            auto capacidad = base::pow2(N);
            auto buffer = polynomial(capacidad);

            buffer.insert(raiz);
            superset_visit(conv.monomial_of(raiz), [&](const auto&
                superconjunto) {
                buffer.insert(conv.index_of(superconjunto));
                return true;
            });

            for (;;) {
                auto indice_lider = buffer.find_leader(capacidad);
                auto& reductor = torre[indice_lider];
                auto esperado = std::uint8_t(impl::INDEFINIDO);

                if (reductor.estado.compare_exchange_strong(esperado,
                    impl::BLOQUEADO)) {
                    reductor.polinomio = sparse_polynomial(buffer,
                        indice_lider + 1);
                    reductor.estado.store(impl::DEFINIDO);
                    return;
                }
                else {
                    while (reductor.estado.load() != impl::DEFINIDO) {
```

```
                continue;
            }

            reductor.polinomio.leaf_visit(
                sparse_exclusive_merge(buffer, indice_lider + 1))
                ;
            capacidad = indice_lider;
        }
    }
});
}
std::cerr << "diagonalizando y simplificando...\n";
tbb::concurrent_vector<std::uint32_t> irreducibles;
tbb::parallel_for(std::uint32_t(0), base::pow2(N), [&](auto
    indice) {
    auto& datos = torre[indice];
    auto existe = [&](const auto& subconjunto) {
        return std::make_pair(!torre[conv.index_of(subconjunto)].
            polinomio.empty(), true);
    };

    if (datos.polinomio.empty() || subset_find(conv.monomial_of
        (indice), existe)) {
        return;
    }

    irreducibles.push_back(indice);

    polynomial buffer(indice + 1);
    datos.polinomio.leaf_visit(sparse_assign(buffer, indice + 1)
        );
    auto cambios = false;

    for (std::uint32_t i = indice; i > 0; --i) {
        auto indice_checar = i - 1;
```

```
            auto& datos_checar = torre[indice_checar];
            if (buffer.find(indice_checar) && !datos_checar.polinomio
               .empty( )) {
               std::shared_lock<std::shared_timed_mutex> sinc(
                  datos_checar.sincronizacion);
               datos_checar.polinomio.leaf_visit(
                  sparse_exclusive_merge(buffer, indice_checar + 1));

               cambios = true;
            }
         }

         if (cambios) {
            std::unique_lock<std::shared_timed_mutex> sinc(datos.
               sincronizacion);
            datos.polinomio = sparse_polynomial(buffer, indice + 1);
         }
      });

      tbb::parallel_sort(irreducibles);

      std::vector<sparse_polynomial> res;
      std::for_each(irreducibles.begin( ), irreducibles.end( ), [&](
         auto indice) {
         res.push_back(std::move(torre[indice].polinomio));
      });

      return res;
   }

   namespace impl {
      struct registro_lista {
         sparse_polynomial prohibidos;
         sparse_polynomial simbolico;
      };
```

```
}

template<std::uint32_t N, typename C>
inline std::vector<sparse_polynomial> metodo_lista(const std::
    vector<char>& tabla, const C& conv)
{
    tbb::concurrent_vector<std::uint32_t> prohibidos;
    tbb::parallel_for(std::uint32_t(0), base::pow2(N), [&](auto i)
        {
        if (!tabla[i]) {
            prohibidos.push_back(i);
        }
    });

    std::vector<sparse_polynomial> base(base::pow2(N));
    std::vector<impl::registro_lista> torre(prohibidos.size());
std::cerr << "procesando con " << prohibidos.size() << "
    prohibidos...\n";
    for (std::uint32_t i = 0; i < base::pow2(N); ++i) {
if (i %1000 == 0) { std::cerr << "\ten " << i << " de " << base::
    pow2(N) << '\n'; }
        auto monomio = conv.monomial_of(i);
        auto divisible = subset_find(monomio, [&](const auto&
            subconjunto) {
            return std::make_pair(!base[conv.index_of(subconjunto)].
                empty(), true);
        });

        if (divisible) {
            continue;
        }

        polynomial simbolico(i + 1);
        simbolico.insert(i);
```

```
    polynomial buffer(prohibidos.size( ));
    for (std::uint32_t j = 0; j < prohibidos.size( ); ++j) {
        if (greatest_common_divisor(conv.monomial_of(prohibidos[j
            ]), monomio) == monomio) {
            buffer.insert(j);
        }
    }

    auto capacidad = prohibidos.size( );

    for (;;) {
        auto indice_lider = buffer.find_leader(capacidad);
        if (indice_lider == std::uint32_t(-1)) {
            base[i] = sparse_polynomial(simbolico, i + 1);
            break;
        }

        auto& reductor = torre[indice_lider];
        if (reductor.prohibidos.empty( )) {
            reductor.prohibidos = sparse_polynomial(buffer,
                indice_lider + 1);
            reductor.simbolico = sparse_polynomial(simbolico, i +
                1);
            break;
        }
        else {
            reductor.prohibidos.leaf_visit(sparse_exclusive_merge(
                buffer, indice_lider + 1));
            reductor.simbolico.leaf_visit(sparse_exclusive_merge(
                simbolico, i + 1));
            capacidad = indice_lider;
        }
    }
}
```

```
    std::vector<sparse_polynomial> res;
    std::for_each(base.begin(), base.end(), [&](auto& polinomio)
        {
        if (!polinomio.empty()) {
            res.push_back(std::move(polinomio));
        }
    });

    return res;
}

int main()
{
    constexpr std::uint32_t N = 9;
//tbb::task_scheduler_init mt(1);
    std::vector<char> tabla(base::pow2(N));
    lex_converter<N> conv;
auto t0 = std::chrono::high_resolution_clock::now();
    std::vector<monomial<N>> trabajo;
    while (get_polynomial(std::cin, trabajo)) {
        evalua_polinomio(trabajo, tabla, conv);
    }

    auto res = (std::accumulate(tabla.begin(), tabla.end(), std::
        uint32_t(0)) < base::pow2(N) / 2 ? metodo_matriz<N>(tabla,
        conv) : metodo_lista<N>(tabla, conv));
auto t1 = std::chrono::high_resolution_clock::now();
std::cerr << std::chrono::duration_cast<std::chrono::milliseconds
    >(t1 - t0).count() / 1000.0 << '\n';
    std::for_each(res.rbegin(), res.rend(), [&](const auto&
        polinomio) {
        trabajo.clear();
        escribe_polinomio(polinomio, std::back_inserter(trabajo),
            conv);
        std::reverse(trabajo.begin(), trabajo.end());
```

```
        write_polynomial(std::cerr, trabajo) << '\n';
    });
}
```

A.2. Detección de instrucciones vectoriales disponibles

```
#ifndef SIMD_H
#define SIMD_H

#include <base/functional.h>
#include <cstddef>
#include <type_traits>
#include <x86intrin.h>

#define CACHE_LINE_SIZE 64

using block_type = unsigned long long;
using vector_type =
    #if defined(__AVX2__)
        __m256i;
    #else
        __m128i;
    #endif

static_assert(sizeof(vector_type) <= CACHE_LINE_SIZE && std::
    is_same<long long __attribute__((vector_size(sizeof(vector_type)
    ))), vector_type>::value, "simd");

template<typename T>
inline bool is_zero(const T& v)
{
    return v == 0;
}

inline bool is_zero(const __m128i& v)
{
```

```
    return _mm_movemask_epi8(_mm_cmpeq_epi32(v, _mm_setzero_si128(
        ))) == 0xFFFF;
}

inline bool is_zero(const __m256i& v)
{
    return _mm256_testz_si256(v, v);
}

#endif
```

A.3. Lectura y escritura de polinomios

```
#ifndef POLYNOMIAL_IO_H
#define POLYNOMIAL_IO_H

#include <base/bits.h>
#include <cstddef>
#include <cctype>
#include <iostream>

template<typename M>
inline std::istream& get_monomial(std::istream& is, M& m)
{
    m.clear();

    if ((is >> std::ws).peek() == '1') {
        return is.ignore();
    }

    for (;;) {
        std::size_t i;

        if (std::tolower((is >> std::ws).peek()) == 'x' && is.
            ignore() >> i && i < m.capacity()) {
            m.insert(i);
```

```
            if (!is.eof( ) && is >> std::ws && !is.eof( ) && is.peek(
                ) == '^') {
               std::size_t e;

               if (is.ignore( ) >> e && e == 0) {
                  m.erase(i);
               }
            }

            if (!is.eof( ) && is >> std::ws && !is.eof( ) && is.peek(
                ) == '*') {
               is.ignore( );
               continue;
            }
         }
         else {
            is.setstate(std::ios_base::failbit);
         }

         break;
      }

      return is;
}

template<typename P>
inline std::istream& get_polynomial(std::istream& is, P& p)
{
      p.clear( );

      if (is >> std::ws && is.peek( ) == '0') {
         return is.ignore( );
      }
```

```
    for (;;) {
        typename P::value_type m;

        if (get_monomial(is, m)) {
            p.push_back(std::move(m));

            if (!is.eof() && is >> std::ws && !is.eof() && is.peek(
                ) == '+') {
                is.ignore();
                continue;
            }
        }
        else {
            is.setstate(std::ios_base::failbit);
        }

        break;
    }

    return is;
}

template<typename M>
inline std::ostream& write_monomial(std::ostream& os, const M& m)
{
    if (m.none()) {
        return os << '1';
    }

    auto tope = m.degree() - 1;

    for (std::size_t i = 0; i < tope; ++i) {
        os << 'x' << m[i] << '*';
    }
```

```
    return os << 'x' << m[tope];
}

template<typename P>
inline std::ostream& write_polynomial(std::ostream& os, const P& p
    )
{
    if (p.empty( )) {
        return os << '0';
    }

    auto tope = p.size( ) - 1;

    for (std::size_t i = 0; i < tope; ++i) {
        write_monomial(os, p[i]) << " + ";
    }

    return write_monomial(os, p[tope]);
}

#endif
```

A.4. Monomios booleanos

```
#ifndef MONOMIAL_H
#define MONOMIAL_H

#include <base/algorithm.h>
#include <base/bits.h>
#include <base/functional.h>
#include <iterator>

template<std::uint32_t N>
class monomial {
public:
    monomial( )
```

```
{
}

static constexpr std::uint32_t capacity( )
{
   return N;
}

std::uint32_t degree( ) const
{
   return bits_.count( );
}

std::uint32_t operator[](std::uint32_t i) const
{
   return bits_.find_ith_set(i);
}

bool none( ) const
{
   return bits_.none( );
}

bool find(std::uint32_t i) const
{
   return bits_[i];
}

void insert(std::uint32_t i)
{
   bits_[i].set( );
}

void erase(std::uint32_t i)
{
```

```
    bits_[i].reset();
}

void clear()
{
    bits_.reset();
}

template<typename F>
void visit(const F& f) const
{
    bits_.visit(f);
}

monomial& operator&=(const monomial& m)
{
    bits_ &= m.bits_;
    return *this;
}

monomial& operator*=(const monomial& m)
{
    bits_ |= m.bits_;
    return *this;
}

monomial& operator/=(const monomial& m)
{
    bits_ ^= m.bits_;
    return *this;
}

monomial operator~() const
{
    auto res = *this;
```

```
        res.bits_ = ~res.bits_;

        return res;
    }

    bool operator==(const monomial& m) const
    {
        return bits_ == m.bits_;
    }

    bool operator!=(const monomial& m) const
    {
        return bits_ != m.bits_;
    }

private:
    base::bitset<N> bits_;
};

template<std::uint32_t N>
inline monomial<N> operator&(monomial<N>&& a, const monomial<N>& b
    )
{
    a &= b;
    return std::move(a);
}

template<std::uint32_t N>
inline monomial<N> operator&(const monomial<N>& a, const monomial<
    N>& b)
{
    return monomial<N>(a) & b;
}

template<std::uint32_t N>
```

```
inline monomial<N> operator*(monomial<N>&& a, const monomial<N>& b
    )
{
    a *= b;
    return std::move(a);
}

template<std::uint32_t N>
inline monomial<N> operator*(const monomial<N>& a, const monomial<
    N>& b)
{
    return monomial<N>(a) * b;
}

template<std::uint32_t N>
inline monomial<N> operator/(monomial<N>&& a, const monomial<N>& b
    )
{
    a /= b;
    return std::move(a);
}

template<std::uint32_t N>
inline monomial<N> operator/(const monomial<N>& a, const monomial<
    N>& b)
{
    return monomial<N>(a) / b;
}

template<std::uint32_t N>
inline monomial<N> least_common_multiple(const monomial<N>& a,
    const monomial<N>& b)
{
    return a * b;
}
```

```
template<std::uint32_t N>
inline monomial<N> greatest_common_divisor(const monomial<N>& a,
    const monomial<N>& b)
{
    return a & b;
}

template<std::uint32_t N>
inline bool is_divisible(const monomial<N>& a, const monomial<N>&
    b)
{
    return least_common_multiple(a, b) == a;
}

template<std::uint32_t N>
inline bool is_coprime(const monomial<N>& a, const monomial<N>& b)
{
    return greatest_common_divisor(a, b).none( );
}

namespace impl {
    template<std::uint32_t N, typename RI, typename F, typename OP>
    inline void set_visit(const monomial<N>& m, RI ai, RI af, const
        F& f, const OP& op)
    {
        for (auto i = ai; i != af; ++i) {
            auto actual = op(m, *i);

            if (f(actual)) {
                set_visit(actual, i + 1, af, f, op);
            }
        }
    }
```

```
template<std::uint32_t N, typename RI, typename F, typename OP>
inline bool set_find(const monomial<N>& m, RI ai, RI af, const
   F& f, const OP& op)
{
   for (auto i = ai; i != af; ++i) {
      auto actual = op(m, *i);
      auto res = f(actual);

      if (res.first || res.second && set_find(actual, i + 1, af
         , f, op)) {
         return true;
      }
   }

   return false;
}

template<std::uint32_t N, typename F, typename V, typename OP>
inline auto set_walk(const monomial<N>& m, const monomial<N>&
   bandera, const F& f, const V& v, const OP& op)
{
   std::array<std::uint32_t, N> prendidos;
   auto escribir = prendidos.begin( );

   bandera.visit([&](auto pos) {
      *escribir++ = pos;
   });

   return v(m, prendidos.begin( ), escribir, f, op);
}

template<std::uint32_t N, typename F, typename V>
inline auto subset_walk(const monomial<N>& m, const F& f, const
    V& v)
{
```

```
        return set_walk(m, m, f, v, [](auto m, auto i) {
            return m.erase(i), m;
        });
    }

    template<std::uint32_t N, typename F, typename V>
    inline auto superset_walk(const monomial<N>& m, const F& f,
        const V& v)
    {
        return set_walk(m, ~m, f, v, [](auto m, auto i) {
            return m.insert(i), m;
        });
    }
}

template<std::uint32_t N, typename F>
inline void subset_visit(const monomial<N>& m, const F& f)
{
    impl::subset_walk(m, f, FUNCTOR(impl::set_visit));
}

template<std::uint32_t N, typename F>
inline bool subset_find(const monomial<N>& m, const F& f)
{
    return impl::subset_walk(m, f, FUNCTOR(impl::set_find));
}

template<std::uint32_t N, typename F>
inline void superset_visit(const monomial<N>& m, const F& f)
{
    impl::superset_walk(m, f, FUNCTOR(impl::set_visit));
}

template<std::uint32_t N, typename F>
inline bool superset_find(const monomial<N>& m, const F& f)
```

```
{
    return impl::superset_walk(m, f, FUNCTOR(impl::set_find));
}

#endif
```

A.5. Orden graduado lexicográfico reverso

```
#ifndef GREVLEX_H
#define GREVLEX_H

#include "monomial.h"
#include <base/array.h>
#include <base/bits.h>
#include <tbb/parallel_for.h>
#include <algorithm>
#include <cstdint>
#include <vector>

struct grevlex {
    template<std::uint32_t N>
    bool operator()(const monomial<N>& a, const monomial<N>& b)
        const
    {
        auto ac = a.degree();
        auto bc = b.degree();

        if (ac != bc) {
            return ac < bc;
        }

        for (std::uint32_t i = ac; i > 0; --i) {
            auto ai = a[i - 1];
            auto bi = b[i - 1];

            if (ai != bi) {
```

```
            return ai > bi;
        }
    }

    return false;
  }
};

template<std::uint32_t N>
class grevlex_converter {
public:
  grevlex_converter()
  {
    for (std::uint32_t f = 0; f <= N; ++f) {
      for (std::uint32_t u = 0; u <= N; ++u) {
        cuenta_[f][u] = (u > f ? 0 : (u == 0 || u == f ? 1 :
          cuenta_[f - 1][u] + cuenta_[f - 1][u - 1]));
      }
    }

    for (std::uint32_t i = 0; i <= N + 1; ++i) {
      acumulado_previo_[i] = (i == 0 ? 0 : acumulado_previo_[i
        - 1] + cuenta_[N][i - 1]);
    }
  }

  std::uint32_t degree_of(std::uint32_t i) const
  {
    return std::upper_bound(acumulado_previo_.begin(),
      acumulado_previo_.end(), i) - acumulado_previo_.begin()
      - 1;
  }

  std::uint32_t degree_of(const monomial<N>& m) const
  {
```

```
        return m.degree( );
    }

    monomial<N> monomial_of(std::uint32_t i) const
    {
        auto grado = degree_of(i);
        return genera_monomio_(inverso_(i - population_before(grado)
            , grado), grado);
    }

    std::uint32_t index_of(const monomial<N>& m) const
    {
        auto grado = degree_of(m);
        return inverso_(genera_indice_(m, grado), grado) +
            population_before(grado);
    }

    std::uint32_t population_before(std::uint32_t d) const
    {
        return acumulado_previo_[d];
    }

    std::uint32_t population_up_to(std::uint32_t d) const
    {
        return acumulado_previo_[d + 1];
    }

    std::uint32_t population_of(std::uint32_t d) const
    {
        return population_up_to(d) - population_before(d);
    }

private:
    std::uint32_t inverso_(std::uint32_t i, std::uint32_t d) const
    {
```

```
    return population_of(d) - i - 1;
}

monomial<N> genera_monomio_(std::uint32_t i, std::uint32_t u)
    const
{
    auto m = monomial<N>();
    auto f = std::uint32_t(N);

    while (i != 0) {
        auto temp = cuenta_[--f][u];

        if (i >= temp) {
            m.insert(f);

            i -= temp;
            u -= 1;
        }
    }

    for (std::uint32_t i = 0; i < u; ++i) {
        m.insert(i);
    }

    return m;
}

std::uint32_t genera_indice_(const monomial<N>& m, std::
    uint32_t d) const
{
    auto i = std::uint32_t(0);
    auto u = std::uint32_t(0);

    m.visit([&](auto pos) {
        i += cuenta_[pos][++u];
```

```
        });

        return i;
    }

    base::matrix<std::uint32_t, N + 1, N + 1> cuenta_;
    std::array<std::uint32_t, N + 2> acumulado_previo_;
};

template<std::uint32_t N>
class grevlex_cache : public grevlex_converter<N> {
public:
    grevlex_cache( )
    {
    }

    grevlex_cache(std::uint32_t d)
    {
        emplace_degree(d);
    }

    auto degree_of(const monomial<N>& m) const
    {
        return static_cast<const grevlex_converter<N>*>(this)->
            degree_of(m);
    }

    auto degree_of(std::uint32_t i) const
    {
        return monomios_[i].degree( );
    }

    auto monomial_of(std::uint32_t i) const
    {
        return monomios_[i];
```

```
    }

    void emplace_degree(std::uint32_t d)
    {
        auto actuales = monomios_.size();
        monomios_.resize(grevlex_converter<N>::population_up_to(d));

        tbb::parallel_for(actuales, monomios_.size(), [this](auto i
            ) {
            monomios_[i] = static_cast<const grevlex_converter<N>*>(
                this)->monomial_of(i);
        });
    }

private:
    std::vector<monomial<N>> monomios_;
};

#endif
```

A.6. Orden lexicográfico

```
#ifndef LEX_H
#define LEX_H

#include "monomial.h"
#include <base/bits.h>
#include <cstdint>

struct lex {
    template<std::uint32_t N>
    bool operator()(const monomial<N>& a, const monomial<N>& b)
        const
    {
        for (std::uint32_t i = 0; i < N; ++i) {
            auto ai = a[i - 1];
```

```
            auto bi = b[i - 1];

            if (ai != bi) {
                return ai < bi;
            }
        }

        return false;
    }
};

template<std::uint32_t N>
class lex_converter {
public:
    std::uint32_t degree_of(std::uint32_t i) const
    {
        return base::popcount(i);
    }

    std::uint32_t degree_of(const monomial<N>& m) const
    {
        return m.degree();
    }

    monomial<N> monomial_of(std::uint32_t i) const
    {
        monomial<N> res;

        while (i != 0) {
            res.insert(N - 1 - base::pop_first_set_bit(i));
        }

        return res;
    }
```

```
    std::uint32_t index_of(const monomial<N>& m) const
    {
        std::uint32_t res = 0;

        m.visit([&](auto i) {
            base::set_bit(res, N - 1 - i);
        });

        return res;
    }

    std::uint32_t population_up_to(std::uint32_t d) const
    {
        return base::pow2(N);
    }
};

#endif
```

A.7. Polinomios booleanos (arreglos de bits)

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include "simd.h"
#include <base/algorithm.h>
#include <base/bits.h>
#include <base/functional.h>
#include <base/integer.h>
#include <tbb/scalable_allocator.h>
#include <algorithm>
#include <climits>
#include <cstdint>
#include <functional>

template<typename T>
```

```
inline bool empty_segment(const T& v)
{
   return is_zero(v);
}

template<typename T>
inline void clear_segment(T& destino)
{
   destino = T{ };
}

template<typename T>
inline void exclusive_merge_segment(T& destino, const T& origen)
{
   destino ^= origen;
}

class polynomial {
public:
   polynomial( ) = default;

   explicit polynomial(std::uint32_t capacidad)
   : mem_(base::make_unique_array<vector_type>(tbb::
      scalable_allocator<vector_type>( ), vectors(capacidad)))
   {
      clear(capacidad);
   }

   static constexpr std::uint32_t bits_per_block( )
   {
      return sizeof(block_type) * CHAR_BIT;
   }

   static constexpr std::uint32_t bits_per_vector( )
   {
```

```
    return sizeof(vector_type) * CHAR_BIT;
}

static std::uint32_t blocks(std::uint32_t capacidad)
{
    return base::ceil_division(capacidad, bits_per_block( ));
}

static std::uint32_t vectors(std::uint32_t capacidad)
{
    return base::ceil_division(capacidad, bits_per_vector( ));
}

bool empty( ) const
{
    return mem_.get( ) == nullptr;
}

void reset( )
{
    mem_.reset( );
}

void clear(std::uint32_t capacidad)
{
    std::for_each(vector_begin( ), vector_end(capacidad),
        FUNCTOR(clear_segment));
}

bool find(std::uint32_t i) const
{
    return base::get_bit(block_begin( )[i / bits_per_block( )],
        i % bits_per_block( ));
}
```

```
void insert(std::uint32_t i)
{
    base::set_bit(block_begin()[i / bits_per_block()], i %
        bits_per_block());
}

void erase(std::uint32_t i)
{
    base::reset_bit(block_begin()[i / bits_per_block()], i %
        bits_per_block());
}

void insert_erase(std::uint32_t i)
{
    base::flip_bit(block_begin()[i / bits_per_block()], i %
        bits_per_block());
}

void exclusive_merge(std::uint32_t capacidad, const polynomial&
    p)
{
    std::transform(vector_begin(), vector_end(capacidad), p.
        vector_begin(), vector_begin(), std::bit_xor<>());
}

std::uint32_t find_leader(std::uint32_t capacidad) const
{
    auto ai = vector_begin();
    auto af = vector_end(capacidad);
    auto aj = base::find_last_if(ai, af, base::not_fn(FUNCTOR(
        empty_segment)));

    if (aj == af) {
        return -1;
    }
```

```
    auto bi = reinterpret_cast<const block_type*>(aj);

    for (std::uint32_t i = bits_per_vector( ) / bits_per_block(
        ); i > 0; --i) {
        auto local = bi[i - 1];

        if (local != 0) {
            return bits_per_vector( ) * std::uint32_t(aj - ai) +
                bits_per_block( ) * std::uint32_t(i - 1) + base::
                find_last_set_bit(local);
        }
    }
}

template<typename F>
void visit(std::uint32_t capacidad, const F& f) const
{
    auto ai = vector_begin( );
    auto af = vector_end(capacidad);
    auto aj = ai;

    while (aj = std::find_if(aj, af, base::not_fn(FUNCTOR(
        empty_segment))), aj != af) {
        auto bi = reinterpret_cast<const block_type*>(aj);

        for (std::uint32_t i = 0; i < bits_per_vector( ) /
            bits_per_block( ); ++i) {
            auto local = bi[i];

            while (local != 0) {
                f(bits_per_vector( ) * std::uint32_t(aj - ai) +
                    bits_per_block( ) * i + base::pop_first_set_bit(
                    local));
            }
```

```
      }

      ++aj;
   }
}

block_type* block_begin( )
{
   return reinterpret_cast<block_type*>(mem_.get( ));
}

block_type* block_end(std::uint32_t capacidad)
{
   return reinterpret_cast<block_type*>(mem_.get( )) + blocks(
      capacidad);
}

const block_type* block_begin( ) const
{
   return reinterpret_cast<const block_type*>(mem_.get( ));
}

const block_type* block_end(std::uint32_t capacidad) const
{
   return reinterpret_cast<const block_type*>(mem_.get( )) +
      blocks(capacidad);
}

vector_type* vector_begin( )
{
   return mem_.get( );
}

vector_type* vector_end(std::uint32_t capacidad)
{
```

```
        return mem_.get( ) + vectors(capacidad);
    }

    const vector_type* vector_begin( ) const
    {
        return mem_.get( );
    }

    const vector_type* vector_end(std::uint32_t capacidad) const
    {
        return mem_.get( ) + vectors(capacidad);
    }

private:
    base::unique_ptr<vector_type[], tbb::scalable_allocator<
        vector_type>> mem_;
};

#endif
```

A.8. Polinomios booleanos (comprimidos)

```
#ifndef SPARSE_POLYNOMIAL_H
#define SPARSE_POLYNOMIAL_H

#include "simd.h"
#include <base/bits.h>
#include <base/integer.h>
#include <base/memory.h>
#include <tbb/scalable_allocator.h>
#include <algorithm>
#include <climits>
#include <cstdint>
#include <iterator>

namespace impl {
```

```
inline constexpr std::pair<std::uint32_t, std::uint32_t>
   memoria_completo(std::uint8_t b)
{
   std::pair<std::uint32_t, std::uint32_t> res(0, 0);

   for (auto i = 1; i <= b; ++i) {
      res.first *= 256;
      res.first += 1;

      res.second *= 256;
      res.second += 32;
   }

   return res;
}

inline constexpr std::pair<std::uint32_t, std::uint32_t>
   memoria_maxima(std::uint32_t v)
{
   auto c4 = base::get_n_bits(v, 24, 8);
   auto c3 = base::get_n_bits(v, 16, 8);
   auto c2 = base::get_n_bits(v,  8, 8);
   auto c1 = base::get_n_bits(v,  0, 8);

   constexpr auto m3 = memoria_completo(3);
   constexpr auto m2 = memoria_completo(2);
   constexpr auto m1 = memoria_completo(1);
   constexpr auto m0 = memoria_completo(0);

   std::pair<std::uint32_t, std::uint32_t> res(0, 0);

   res.first += c4 * m3.first + 1;
   res.first += c3 * m2.first + 1;
   res.first += c2 * m1.first + 1;
   res.first += c1 * m0.first + 1;
```

```
    res.second += c4 * m3.second + std::min(c4 + 1, std::
        uint32_t(32));
    res.second += c3 * m2.second + std::min(c3 + 1, std::
        uint32_t(32));
    res.second += c2 * m1.second + std::min(c2 + 1, std::
        uint32_t(32));
    res.second += c1 * m0.second + std::min(c1 + 1, std::
        uint32_t(32));

    return { res.first, res.second + base::multiple_underflow(
        res.second, 32) };
}


static_assert(base::is_little_endian(), "sparse_polynomial");

inline void escribe_bits(std::pair<std::uint8_t*, std::uint8_t
    >& datos1, std::uint8_t v)
{
    *reinterpret_cast<std::uint16_t*>(datos1.first) |= std::
        uint16_t(v) << datos1.second;

    datos1.second += 5;
    datos1.first += (datos1.second >= 8);
    datos1.second %= 8;
}

inline auto lee_bits(std::pair<const std::uint8_t*, std::
    uint8_t>& datos1)
{
    auto res = base::get_n_bits(*reinterpret_cast<const std::
        uint16_t*>(datos1.first), datos1.second, 5);

    datos1.second += 5;
```

```
    datos1.first += (datos1.second >= 8);
    datos1.second %= 8;

    return res;
}


inline std::uint8_t* comprime_headers(std::uint8_t* ai, std::
    uint8_t* af)
{
    std::pair<std::uint8_t*, std::uint8_t> res(ai, 0);

    for (; ai != af; ++ai) {
        auto actual = *ai;
        base::reset_bits(*ai);

        escribe_bits(res, actual);
    }

    return res.first + bool(res.second);
}

inline bool procesa_bitset(std::reverse_iterator<std::uint8_t
    *>& datos1, std::reverse_iterator<std::uint8_t*>& datos2, std
    ::reverse_iterator<std::uint8_t*>& datos3, const base::bitset
    <256, block_type>& v)
{
    auto cuantos = std::uint16_t(v.count( ));

    if (cuantos != 0) {
        *datos1++ = cuantos * (cuantos < 32);

        if (cuantos < 32) {
            datos2 += cuantos;
            auto copia = datos2.base( );
```

```
            v.visit([&](auto i) {
                *copia++ = i;
            });
        }
        else {
            datos3 += 32;
            auto& bitset = *reinterpret_cast<base::bitset<256,
                block_type>*>(datos3.base( ));

            bitset = v;
        }
    }

    return cuantos;
}

inline std::uint8_t* comprime(std::reverse_iterator<std::
    uint8_t*>& datos1, std::reverse_iterator<std::uint8_t*>&
    datos2, std::reverse_iterator<std::uint8_t*>& datos3, const
    block_type* pi, const block_type* pf)
{
    constexpr auto bits_per_block = sizeof(block_type) *
        CHAR_BIT;

    auto datos1_ini = datos1;
    auto offset = std::uint32_t(pf - pi) * bits_per_block;

    base::bitset<256, block_type> v1;
    base::bitset<256, block_type> v2;
    base::bitset<256, block_type> v3;
    base::bitset<256, block_type> v4;

    while (offset != 0) {
        offset -= bits_per_block;
```

```
        v1.block_begin( )[(offset %256) / bits_per_block] = *--
            pf;

        if (offset %256 == 0) {
            if (procesa_bitset(datos1, datos2, datos3, v1)) {
                v1.reset( );
                v2[base::get_n_bits(offset, 8, 8)].set( );
            }

            if (offset %65536 == 0) {
                if (procesa_bitset(datos1, datos2, datos3, v2)) {
                    v2.reset( );
                    v3[base::get_n_bits(offset, 16, 8)].set( );
                }

                if (offset %16777216 == 0) {
                    if (procesa_bitset(datos1, datos2, datos3, v3))
                        {
                        v3.reset( );
                        v4[base::get_n_bits(offset, 24, 8)].set( );
                    }
                }
            }
        }
    }

    procesa_bitset(datos1, datos2, datos3, v4);
    return comprime_headers(datos1.base( ), datos1_ini.base( ));
}


template<std::uint8_t B>
struct descomprime_impl {
    template<typename F>
    void operator()(std::pair<const std::uint8_t*, std::uint8_t
```

```
        >& datos1, const std::uint8_t*& datos2, const std::uint8_t
        *& datos3, const F& f, std::uint32_t prefijo)
    {
        auto cuantos = impl::lee_bits(datos1);

        if (cuantos != 0) {
            auto copia = datos2;
            datos2 += cuantos;

            for (auto i = 0; i < cuantos; ++i) {
                descomprime_impl<B - 1>()(datos1, datos2, datos3,
                    f, prefijo | std::uint32_t(copia[i]) << (8 * (B -
                     1)));
            }
        }
        else {
            auto& bitset = *reinterpret_cast<const base::bitset
                <256, block_type>*>(datos3);
            datos3 += 32;

            bitset.visit([&](auto i) {
                descomprime_impl<B - 1>()(datos1, datos2, datos3,
                    f, prefijo | std::uint32_t(i) << (8 * (B - 1)));
            });
        }
    }
};

template<>
struct descomprime_impl<1> {
    template<typename F>
    void operator()(std::pair<const std::uint8_t*, std::uint8_t
        >& datos1, const std::uint8_t*& datos2, const std::uint8_t
        *& datos3, const F& f, std::uint32_t prefijo)
    {
```

```
        auto cuantos = impl::lee_bits(datos1);

        if (cuantos != 0) {
            auto copia = datos2;
            datos2 += cuantos;

            f(prefijo, copia, datos2);
        }
        else {
            auto& bitset = *reinterpret_cast<const base::bitset
                <256, block_type>*>(datos3);
            datos3 += 32;

            f(prefijo, bitset);
        }
    }
};

template<typename F>
inline void descomprime(const std::uint8_t* d1, const std::
    uint8_t* datos2, const std::uint8_t* datos3, const F& f)
{
    std::pair<const std::uint8_t*, std::uint8_t> datos1(d1, 0);
    descomprime_impl<4>()(datos1, datos2, datos3, f, 0);
}


template<typename F>
struct visitante {
    void operator()(std::uint32_t prefijo, const base::bitset
        <256, block_type>& b) const
    {
        b.visit([&, this](auto i) {
            f(prefijo | i);
        });
```

```
        }

        void operator()(std::uint32_t prefijo, const std::uint8_t*
            ini, const std::uint8_t* fin) const
        {
            while (ini != fin) {
                f(prefijo | *ini++);
            }
        }

        const F& f;
    };
}

class sparse_assign {
public:
        sparse_assign(polynomial& p, std::uint32_t c)
        : p_(p), capacidad_(c)
        {
        }

    void operator()(std::uint32_t valor) const
    {
        p_.insert_erase(valor);
    }

    void operator()(std::uint32_t prefijo, const std::uint8_t* ini,
         const std::uint8_t* fin) const
    {
        while (ini != fin) {
            p_.insert(prefijo | *ini++);
        }
    }

    void operator()(std::uint32_t prefijo, const base::bitset<256,
```

```
      block_type>& b) const
   {
      auto pi = p_.block_begin( ) + prefijo / p_.bits_per_block( )
         ;
      auto pf = p_.block_begin( ) + prefijo / p_.bits_per_block( )
          + 256 / p_.bits_per_block( );

      std::copy_n(b.block_begin( ), std::min(pf, p_.block_end(
         capacidad_)) - pi, pi);
   }

private:
   polynomial& p_;
   std::uint32_t capacidad_;
};

class sparse_exclusive_merge {
public:
        sparse_exclusive_merge(polynomial& p, std::uint32_t c)
        : p_(p), capacidad_(c)
        {
        }

   void operator()(std::uint32_t valor) const
   {
      p_.insert_erase(valor);
   }

   void operator()(std::uint32_t prefijo, const std::uint8_t* ini,
       const std::uint8_t* fin) const
   {
      while (ini != fin) {
         p_.insert_erase(prefijo | *ini++);
      }
   }
```

```
    void operator()(std::uint32_t prefijo, const base::bitset<256,
        block_type>& b) const
    {
        auto pi = p_.block_begin() + prefijo / p_.bits_per_block()
            ;
        auto pf = p_.block_begin() + prefijo / p_.bits_per_block()
            + 256 / p_.bits_per_block();

        std::transform(pi, std::min(pf, p_.block_end(capacidad_)), b
            .block_begin(), pi, std::bit_xor<>());
    }

private:
    polynomial& p_;
    std::uint32_t capacidad_;
};

class sparse_polynomial {
public:
    sparse_polynomial() = default;
    sparse_polynomial(const polynomial& p, std::uint32_t capacidad)
    {
        auto mem = impl::memoria_maxima(capacidad);
        alignas(block_type) std::uint8_t memoria_h[mem.first];
        alignas(block_type) std::uint8_t memoria_c[mem.second];
        alignas(block_type) std::uint8_t memoria_v[mem.second];

        auto datos1 = std::make_reverse_iterator(memoria_h + mem.
            first);
        auto datos2 = std::make_reverse_iterator(memoria_c + mem.
            second);
        auto datos3 = std::make_reverse_iterator(memoria_v + mem.
            second);
        auto fin_headers = impl::comprime(datos1, datos2, datos3, p.
```

```
        block_begin( ), p.block_end(capacidad));

    auto tam_header = datos1 - std::make_reverse_iterator(
        fin_headers);
    auto tam_octeto = datos2 - std::make_reverse_iterator(
        memoria_c + mem.second);
    auto tam_vector = datos3 - std::make_reverse_iterator(
        memoria_v + mem.second);
    mem_ = base::make_unique_array<block_type>(tbb::
        scalable_allocator<block_type>( ), base::ceil_division(
        offset_( ) + tam_header + tam_octeto + tam_vector, sizeof(
        block_type)));
    guarda_datos_(tam_header, tam_octeto, tam_vector);

    auto info = buffer_hcv_( );
    std::copy(datos1.base( ), fin_headers, std::get<0>(info));
    std::copy(datos2.base( ), memoria_c + mem.second, std::get
        <1>(info));
    std::copy(datos3.base( ), memoria_v + mem.second, std::get
        <2>(info));
}

bool empty( ) const
{
    return mem_.get( ) == nullptr;
}

void reset( )
{
    mem_.reset( );
}

template<typename F>
void visit(const F& f) const
{
```

```
        auto info = buffer_hcv_( );
        impl::descomprime(std::get<0>(info), std::get<1>(info), std
            ::get<2>(info), impl::visitante<F>{f});
    }

    template<typename F>
    void leaf_visit(const F& f) const
    {
        auto info = buffer_hcv_( );
        impl::descomprime(std::get<0>(info), std::get<1>(info), std
            ::get<2>(info), f);
    }

private:
    static constexpr std::uint32_t offset_( )
    {
        return std::max(2 * sizeof(std::uint32_t), sizeof(block_type
            ));
    }

    void guarda_datos_(std::uint32_t tam_header, std::uint32_t
        tam_octeto, std::uint32_t tam_vector)
    {
        reinterpret_cast<std::uint32_t*>(mem_.get( ))[0] =
            tam_vector;
        reinterpret_cast<std::uint32_t*>(mem_.get( ))[1] =
            tam_octeto;
    }

    std::tuple<std::uint8_t*, std::uint8_t*, std::uint8_t*>
        buffer_hcv_( ) const
    {
        auto tam_vector = reinterpret_cast<const std::uint32_t*>(
            mem_.get( ))[0];
        auto tam_octeto = reinterpret_cast<const std::uint32_t*>(
```

```
            mem_.get( ))[1];
        auto buffer = reinterpret_cast<std::uint8_t*>(mem_.get( )) +
            offset_( );

        return std::make_tuple(buffer + tam_vector + tam_octeto,
            buffer + tam_vector, buffer);
    }

    base::unique_ptr<block_type[], tbb::scalable_allocator<
        block_type>> mem_;
};

#endif
```

A.9. Variante del algoritmo F4

```
//#define TBB_USE_DEBUG 1
//#define TBB_USE_THREADING_TOOLS 1

#include <tbb/task_scheduler_init.h>
#include <bitset>
#include <fstream>
#include <iostream>
#include <map>
#include <set>
#include <sstream>

#include "grevlex.h"
#include "monomial.h"
#include "polynomial.h"
#include "polynomial_io.h"
#include "sparse_polynomial.h"
#include <base/functional.h>
#include <tbb/concurrent_vector.h>
#include <tbb/parallel_for.h>
#include <tbb/parallel_for_each.h>
```

```
#include <tbb/parallel_sort.h>
#include <tbb/partitioner.h>
#include <tbb/task.h>
#include <algorithm>
#include <array>
#include <atomic>
#include <chrono>
#include <cmath>
#include <cstddef>
#include <cstdint>
#include <iostream>
#include <shared_mutex>
#include <utility>
#include <tuple>

template<std::uint32_t N, typename II, typename P>
inline void lee_polinomio(P& p, II ai, II af, const grevlex_cache<
    N>& conv)
{
    while (ai != af) {
        p.insert_erase(conv.index_of(*ai++));
    }
}

template<std::uint32_t N, typename OI, typename P>
inline OI escribe_polinomio(const P& p, OI oi, const grevlex_cache
    <N>& conv)
{
    p.visit([&](auto i) {
        *oi++ = conv.monomial_of(i);
    });

    return oi;
}
```

```
template<typename F>
inline void visita_disperso(const sparse_polynomial& origen, F&& f
    )
{
    origen.leaf_visit(f);
}

template<typename F, std::uint32_t N>
inline void visita_producto_disperso(const sparse_polynomial&
    origen, const monomial<N>& factor, F&& f, const grevlex_cache<N
    >& conv)
{
    return (factor.none( ) ? visita_disperso(origen, std::forward<F
        >(f)) : origen.visit([&](auto indice) {
        f(conv.index_of(factor * conv.monomial_of(indice)));
    }));
}

template<std::uint32_t N>
inline monomial<N> genera_factor(std::uint32_t i)
{
    monomial<N> res;
    res.insert(i);

    return res;
}

template<std::uint32_t N>
class piramide_polinomial {
    struct registro {
        registro( )
        : reductor(-1)
        {
        }
```

```
        std::uint32_t clave : 31;
        std::uint32_t denso : 1;
        std::uint32_t reductor;
        sparse_polynomial polinomio;
    };

public:
    struct reducible {
        reducible(std::uint32_t n)
        : polinomio(n), indice_lider(-1)
        {
        }

        polynomial polinomio;
        std::uint32_t indice_lider;
    };

    struct reductor {
        reductor(std::uint32_t n)
        : polinomio(n), lista(base::dynamic_create_array<std::
            uint32_t>(std::allocator<std::uint32_t>(), list_capacity(
            n))), fin_lista(nullptr), representante(nullptr)
        {
        }

        reductor(reductor&& r)
        : polinomio(std::move(r.polinomio)), lista(std::move(r.lista
            )), fin_lista(r.fin_lista.load(std::memory_order_relaxed))
            , representante(r.representante.load(std::
            memory_order_relaxed))
        {
        }

        static std::uint32_t list_capacity(std::uint32_t n)
        {
```

```
        return base::ceil_division(n, CHAR_BIT * (sizeof(std::
            uint32_t) + sizeof(vector_type)));
    }

    polynomial polinomio;
    std::unique_ptr<std::uint32_t[]> lista;
    std::atomic<std::uint32_t*> fin_lista;
    std::atomic<const polynomial*> representante;
};

piramide_polinomial( )
: clave_(0), grado_(-1)
{
}

std::uint32_t degree( ) const
{
    return grado_;
}

std::uint32_t size( ) const
{
    return datos_.size( );
}

std::pair<std::uint32_t, bool> get_status(std::uint32_t i)
    const
{
    return { datos_[i].clave, datos_[i].reductor == i };
}

const sparse_polynomial& operator[](std::uint32_t i) const
{
    return datos_[i].polinomio;
}
```

```
    void increase_degree(const grevlex_cache<N>& conv)
    {
        datos_.resize(conv.population_up_to(++grado_));
    }

    void preprocess(tbb::concurrent_vector<std::pair<std::uint32_t,
         std::uint32_t>>& multiplos_pendientes, const grevlex_cache<N
        >& conv)
    {
        tbb::parallel_sort(multiplos_pendientes.begin(),
            multiplos_pendientes.end());
        multiplos_pendientes.resize(std::unique(multiplos_pendientes
            .begin(), multiplos_pendientes.end()) -
            multiplos_pendientes.begin());
        multiplos_pendientes.resize(std::remove_if(
            multiplos_pendientes.begin(), multiplos_pendientes.end()
            , [](const auto& producto) {
            return producto.second == 0;
        }) - multiplos_pendientes.begin());
std::cerr << multiplos_pendientes.size() << " multiplos
    pendientes sin duplicados\n";
        tbb::parallel_for(std::uint32_t(0), std::uint32_t(datos_.
            size()), [&, this](auto indice) {
            auto& datos = datos_[indice];
            datos.reductor = (datos.polinomio.empty() ? std::
                uint32_t(-1) : indice);
            datos.denso &= (datos.reductor != std::uint32_t(-1));
        });

        tbb::concurrent_vector<std::pair<std::uint32_t, std::
            uint32_t>> multiplos_finales;
        std::for_each(multiplos_pendientes.begin(),
            multiplos_pendientes.end(), [&, this](const auto&
            producto) {
```

```
            auto& datos_multiplo = datos_[conv.index_of(
               least_common_multiple(conv.monomial_of(producto.first),
                conv.monomial_of(producto.second)))];
            if (datos_multiplo.reductor == std::uint32_t(-1)) {
               datos_multiplo.reductor = producto.first;
            }
            else {
               multiplos_finales.push_back(producto);
            }
         });
std::cerr << multiplos_finales.size() << " multiplos pendientes
   finales\n";
         multiplos_pendientes = std::move(multiplos_finales);
std::cerr << "a preparar los reductores de las filas faltantes\n";
         tbb::parallel_for(std::uint32_t(0), std::uint32_t(datos_.
            size()), [&, this](auto indice) {
            auto& datos = datos_[indice];
            if (datos.reductor == std::uint32_t(-1)) {
               datos.reductor = this->busca_reductor_(indice, conv);
            }
         });
   }

   template<typename RI1, typename RI2>
   void import(RI1 pi, RI1 pf, RI2 ri, RI2 rf, tbb::
      concurrent_vector<std::uint32_t>& indices, const
      grevlex_cache<N>& conv)
   {
      auto reducibles = base::make_range(pi, pf);
      auto reductores = base::make_range(ri, rf);
      auto fin_actual = std::uint32_t(datos_.size());

      tbb::parallel_for_each(reductores, [&, this](auto& reductor)
          {
         reductor.polinomio.clear(datos_.size());
```

```
            reductor.representante.store(nullptr, std::
                memory_order_relaxed);
        });

        while (fin_actual != 0) {
            auto ini_actual = fin_actual - std::min(fin_actual, std::
                uint32_t(reductores.size( )));

            tbb::parallel_for(ini_actual, fin_actual, [&, this](auto
                indice) {
                auto& datos = datos_[indice];
                auto& reductor = reductores[indice - ini_actual];

                if (reductor.representante.load(std::
                    memory_order_relaxed) == &reductor.polinomio) {
                    if (reductor.fin_lista.load(std::
                        memory_order_relaxed) == nullptr) {
                        reductor.polinomio.clear(indice + 1);
                    }
                    else {
                        for (auto i : base::make_range(reductor.lista.
                            get( ), reductor.fin_lista.load(std::
                            memory_order_relaxed))) {
                            clear_segment(reductor.polinomio.vector_begin
                                ( )[i]);
                        }
                    }
                }

                if (datos.reductor == std::uint32_t(-1)) {
                    reductor.representante.store(nullptr, std::
                        memory_order_relaxed);
                    reductor.fin_lista.store(nullptr, std::
                        memory_order_relaxed);
                }
```

```
        else {
            visita_producto_disperso(datos_[datos.reductor].
                polinomio, conv.monomial_of(indice) / conv.
                monomial_of(datos.reductor), sparse_assign{
                reductor.polinomio, indice + 1}, conv);
            reductor.representante.store(&reductor.polinomio,
                std::memory_order_relaxed);

            if (datos.denso) {
                reductor.fin_lista.store(nullptr, std::
                    memory_order_relaxed);
            }
            else {
                reductor.fin_lista.store(this->escribe_lista_(
                    reductor.polinomio, indice + 1, reductor.lista
                    .get()), std::memory_order_relaxed);
                datos.denso = (reductor.fin_lista.load(std::
                    memory_order_relaxed) == nullptr);
            }
        }
    });

    tbb::parallel_for_each(reducibles, [&, this](auto&
        reducir) {
        while (reducir.indice_lider >= ini_actual && reducir.
            indice_lider < fin_actual) {
            auto& datos_reducir = datos_[reducir.indice_lider];
            auto& reductor = reductores[reducir.indice_lider -
                ini_actual];
            const polynomial* esperado = nullptr;

            if (reductor.representante.compare_exchange_strong(
                esperado, &reducir.polinomio)) {
                reductor.fin_lista.store(this->escribe_lista_(
                    reducir.polinomio, reducir.indice_lider + 1,
```

```
                reductor.lista.get( ));
            indices.push_back(reducir.indice_lider);

            datos_reducir.clave = clave_.fetch_add(1);
            datos_reducir.denso = (reductor.fin_lista.load(
                std::memory_order_relaxed) == nullptr);
            datos_reducir.reductor = reducir.indice_lider;
            datos_reducir.polinomio = sparse_polynomial(
                reducir.polinomio, reducir.indice_lider + 1);

            return;
        }

        if (reductor.fin_lista.load( ) == nullptr) {
            reducir.polinomio.exclusive_merge(reducir.
                indice_lider + 1, *reductor.representante.load
                (std::memory_order_relaxed));
        }
        else {
            for (auto i : base::make_range(reductor.lista.
                get( ), reductor.fin_lista.load(std::
                memory_order_relaxed))) {
                exclusive_merge_segment(reducir.polinomio.
                    vector_begin( )[i], reductor.representante.
                    load(std::memory_order_relaxed)->
                    vector_begin( )[i]);
            }
        }

        reducir.indice_lider = reducir.polinomio.
            find_leader(reducir.indice_lider);
    }
});

fin_actual = ini_actual;
```

```
    }

    tbb::parallel_for_each(reducibles, [&](auto& reducir) {
        reducir.polinomio.clear(reducir.indice_lider + 1);
    });
}

void postprocess(const tbb::concurrent_vector<std::uint32_t>&
    indices, tbb::concurrent_vector<std::uint32_t>&
    indices_eliminar, const grevlex_cache<N>& conv)
{
    tbb::parallel_for_each(indices_eliminar, [&, this](auto
        indice) {
        datos_[indice].polinomio.reset();
    });

    indices_eliminar.clear();

    tbb::parallel_for_each(indices, [&, this](auto indice) {
        datos_[indice].reductor = std::min(indice, this->
            busca_reductor_(indice, conv));
    });
}

void diagonalize(const tbb::concurrent_vector<std::uint32_t>&
    indices, const grevlex_cache<N>& conv)
{
    tbb::parallel_for(std::uint32_t(0), std::uint32_t(datos_.
        size()), [&, this](auto indice) {
        auto& datos = datos_[indice];
        if (datos.polinomio.empty()) {
            datos.reductor = this->busca_reductor_(indice, conv);
        }
    });
```

```
std::vector<std::shared_timed_mutex> sincronizacion(datos_.
    size());
tbb::parallel_for_each(indices, [&, this](auto indice) {
    auto& datos = datos_[indice];
    if (datos.polinomio.empty()) {
        return;
    }

    polynomial buffer(indice + 1);
    visita_disperso(datos.polinomio, sparse_assign{buffer,
        indice + 1});
    auto cambios = false;

    for (std::uint32_t j = indice; j > 0; --j) {
        auto indice_interno = j - 1;
        auto& datos_interno = datos_[indice_interno];
        if (buffer.find(indice_interno) && datos_interno.
            reductor != std::uint32_t(-1)) {
            std::shared_lock<std::shared_timed_mutex> sinc(
                sincronizacion[datos_interno.reductor]);
            visita_producto_disperso(datos_[datos_interno.
                reductor].polinomio, conv.monomial_of(
                indice_interno) / conv.monomial_of(datos_interno.
                reductor), sparse_exclusive_merge{buffer,
                indice_interno + 1}, conv);

            cambios = true;
        }
    }

    if (cambios) {
        std::unique_lock<std::shared_timed_mutex> sinc(
            sincronizacion[indice]);
        datos.polinomio = sparse_polynomial(buffer, indice +
            1);
```

```
        }
    });
}

tbb::concurrent_vector<std::pair<sparse_polynomial, std::
    uint32_t>> extract_up_to(std::uint32_t grado, const
    grevlex_cache<N>& conv)
{
    tbb::concurrent_vector<std::pair<sparse_polynomial, std::
        uint32_t>> res;
    tbb::parallel_for(std::uint32_t(0), conv.population_up_to(
        grado), [&, this](auto indice) {
        auto& datos = datos_[indice];
        if (!datos.polinomio.empty()) {
            res.emplace_back(std::move(datos.polinomio), indice);
        }
    });

    datos_.clear();
    grado_ = std::uint32_t(-1);

    return res;
}

private:
    std::uint32_t busca_reductor_(std::uint32_t indice, const
        grevlex_cache<N>& conv)
    {
        std::uint32_t res = -1;
        subset_visit(conv.monomial_of(indice), [&, this](const auto&
            subconjunto) {
            auto indice_subconjunto = conv.index_of(subconjunto);
            if (!datos_[indice_subconjunto].polinomio.empty()) {
                res = std::min(res, indice_subconjunto);
            }
```

```
            return true;
        });

        return res;
    }

    std::uint32_t* escribe_lista_(const polynomial& polinomio, std
        ::uint32_t capacidad, std::uint32_t* lista)
    {
        auto limite = lista + reductor::list_capacity(capacidad);
        auto vectores = polinomio.vectors(capacidad);

        for (std::uint32_t i = 0; i < vectores; ++i) {
            if (!empty_segment(polinomio.vector_begin( )[i])) {
                if (lista == limite) {
                    return nullptr;
                }

                *lista++ = i;
            }
        }

        return lista;
    }

    std::uint32_t grado_;
    std::atomic<std::uint32_t> clave_;
    std::vector<registro> datos_;
};

namespace impl {
    template<std::uint32_t N>
    inline void reporta_estado(const piramide_polinomial<N>&
        piramide, const grevlex_cache<N>& conv)
```

```
{
    std::array<std::atomic<std::uint32_t>, N + 1> stats = { };
    tbb::parallel_for(std::uint32_t(0), conv.population_up_to(
        piramide.degree( )), [&](auto i) {
        stats[conv.degree_of(i)] += !piramide[i].empty( );
    });

    for (std::uint32_t i = 0; i <= piramide.degree( ); ++i) {
        std::cerr << i << ": " << stats[i] << '\n';
    }

    std::cerr << '\n';
}

template<std::uint32_t N>
inline void descomprime(std::atomic<typename
    piramide_polinomial<N>::reducible*>* escribir, const std::
    pair<sparse_polynomial, std::uint32_t>& comprimido, const
    piramide_polinomial<N>& piramide, const grevlex_cache<N>&
    conv)
{
    auto actual = escribir->fetch_add(1);
    visita_disperso(comprimido.first, sparse_assign{actual->
        polinomio, conv.population_up_to(piramide.degree( ))});
    actual->indice_lider = comprimido.second;
}

template<std::uint32_t N>
inline void multiplica(std::atomic<typename piramide_polinomial
    <N>::reducible*>* escribir, const std::pair<std::uint32_t,
    std::uint32_t>& producto, const piramide_polinomial<N>&
    piramide, const grevlex_cache<N>& conv)
{
    auto actual = escribir->fetch_add(1);
    visita_producto_disperso(piramide[producto.first], conv.
```

```
            monomial_of(producto.second), sparse_assign{actual->
            polinomio, conv.population_up_to(piramide.degree( ))},
            conv);
        actual->indice_lider = actual->polinomio.find_leader(conv.
            population_up_to(piramide.degree( )));
    }

    template<typename T>
    inline void elimina_atras(std::vector<T>& v, std::uint32_t n)
    {
        v.erase(v.end( ) - n, v.end( ));
    }

    template<typename T>
    inline void elimina_atras(tbb::concurrent_vector<T>& v, std::
        uint32_t n)
    {
        v.resize(v.size( ) - n);
    }
}

template<std::uint32_t N>
inline void actualiza_base(const tbb::concurrent_vector<std::pair<
    sparse_polynomial, std::uint32_t>>& polinomios_pendientes, tbb::
    concurrent_vector<std::pair<std::uint32_t, std::uint32_t>>&
    inducidos_pendientes, tbb::concurrent_vector<std::pair<std::
    uint32_t, std::uint32_t>>& multiplos_pendientes, tbb::
    concurrent_vector<std::uint32_t>& indices, tbb::
    concurrent_vector<std::uint32_t>& indices_eliminar, std::
    uint32_t& lineales, piramide_polinomial<N>& piramide, const
    grevlex_cache<N>& conv)
{
std::cerr << "preprocesando...\n";
    std::uint32_t anteriores = indices.size( );
    piramide.preprocess(multiplos_pendientes, conv);
```

```
    std::vector<std::reference_wrapper<const std::pair<
        sparse_polynomial, std::uint32_t>>> entrada_pendientes(
        polinomios_pendientes.begin(), polinomios_pendientes.end())
        ;
std::cerr << "en␣total␣" << entrada_pendientes.size() +
    inducidos_pendientes.size() + multiplos_pendientes.size() << "
    ␣polinomios␣por␣importar...\n";
    auto grupo_reducibles = std::min(std::size_t(2048),
        entrada_pendientes.size() + inducidos_pendientes.size() +
        multiplos_pendientes.size());
    auto grupo_reductores = std::min(std::size_t(48), std::size_t(
        piramide.size()));

    std::vector<typename piramide_polinomial<N>::reducible>
        reducibles;
    for (std::uint32_t i = 0; i < grupo_reducibles; ++i) {
        reducibles.emplace_back(conv.population_up_to(piramide.
            degree()));
    }

    std::vector<typename piramide_polinomial<N>::reductor>
        reductores;
    for (std::uint32_t i = 0; i < grupo_reductores; ++i) {
        reductores.emplace_back(conv.population_up_to(piramide.
            degree()));
    }

    while (!entrada_pendientes.empty() || !inducidos_pendientes.
        empty() || !multiplos_pendientes.empty()) {
        std::atomic<typename piramide_polinomial<N>::reducible*>
            escribir(reducibles.data());
        auto prepara_reducibles = [&](auto& pendientes, const auto&
            func, const auto&... parametros) {
            auto cuantos = std::min(grupo_reducibles - (escribir.load
                (std::memory_order_relaxed) - reducibles.data()),
```

```
            pendientes.size( ));
        tbb::parallel_for_each(pendientes.end( ) - cuantos,
            pendientes.end( ), std::bind(func, &escribir, std::
            placeholders::_1, std::cref(parametros)...));
        impl::elimina_atras(pendientes, cuantos);
      };
impl::reporta_estado(piramide, conv);
      prepara_reducibles(entrada_pendientes, FUNCTOR(impl::
        descomprime<N>), piramide, conv);
      prepara_reducibles(inducidos_pendientes, FUNCTOR(impl::
        multiplica<N>), piramide, conv);
      prepara_reducibles(multiplos_pendientes, FUNCTOR(impl::
        multiplica<N>), piramide, conv);
std::cerr << "importando " << escribir.load(std::
  memory_order_relaxed) - reducibles.data( ) << ", restan " <<
  entrada_pendientes.size( ) + inducidos_pendientes.size( ) +
  multiplos_pendientes.size( ) << '\n';
auto t0 = std::chrono::high_resolution_clock::now( );
      piramide.import(reducibles.data( ), escribir.load(std::
        memory_order_relaxed), reductores.begin( ), reductores.end
        ( ), indices, conv);
auto t1 = std::chrono::high_resolution_clock::now( );
auto total = std::chrono::duration_cast<std::chrono::milliseconds
  >(t1 - t0).count( ) / 1000.0;
std::cerr << total / (escribir.load(std::memory_order_relaxed) -
  reducibles.data( )) << " segundos por reducible, total = " <<
  total << '\n';
  }

  piramide.postprocess(indices, indices_eliminar, conv);
  lineales += std::count_if(indices.begin( ) + anteriores,
    indices.end( ), [&](auto indice) {
    return conv.degree_of(indice) <= 1;
  });
std::cerr << lineales << " lineales actualmente\n\n";
```

```
impl::reporta_estado(piramide, conv);
}

template<std::uint32_t N>
inline void simplifica_indices(tbb::concurrent_vector<std::
    uint32_t>& indices, tbb::concurrent_vector<std::uint32_t>&
    indices_eliminar, std::uint32_t& existentes, const
    piramide_polinomial<N>& piramide)
{
std::cerr << "a simplificar " << indices.size() << " indices\n";
    auto indices_simplificados = tbb::concurrent_vector<std::
        uint32_t>();
    auto indices_eliminables = tbb::concurrent_vector<std::uint32_t
        >();
    auto translado = [&](auto indice) {
        if (piramide.get_status(indice).second) {
            indices_simplificados.push_back(indice);
        }
        else {
            indices_eliminables.push_back(indice);
        }
    };

    tbb::parallel_for_each(indices.begin(), indices.begin() +
        existentes, translado);
    auto existentes_simplificados = indices_simplificados.size();
    tbb::parallel_for_each(indices.begin() + existentes, indices.
        end(), translado);
std::cerr << "quedaron " << indices_simplificados.size() << "
    indices\n";
    existentes = existentes_simplificados;
    indices = std::move(indices_simplificados);
    indices_eliminar = std::move(indices_eliminables);
}
```

```
template<std::uint32_t N>
inline bool criterio_buchberger(const monomial<N>& mcm, const
    monomial<N>& a, std::uint32_t clave_a, const monomial<N>& b, std
    ::uint32_t clave_b, const piramide_polinomial<N>& piramide,
    const grevlex_cache<N>& conv)
{
    auto datos_ab = std::make_tuple(true, mcm.degree(), true, std
        ::max(clave_a, clave_b), std::min(clave_a, clave_b));

    return subset_find(mcm, [&](const auto& c) {
        auto indice_c = conv.index_of(c);

        if (indice_c < piramide.size()) {
            auto estado_c = piramide.get_status(indice_c);
            auto clave_c = estado_c.first;

            if (estado_c.second && clave_a != clave_c && clave_b !=
                clave_c) {
                auto datos_ac = std::make_tuple(!is_coprime(a, c),
                    least_common_multiple(a, c).degree(), !is_divisible
                    (b, c), std::max(clave_a, clave_c), std::min(clave_a
                    , clave_c));
                auto datos_bc = std::make_tuple(!is_coprime(b, c),
                    least_common_multiple(b, c).degree(), !is_divisible
                    (a, c), std::max(clave_b, clave_c), std::min(clave_b
                    , clave_c));

                return std::make_pair(datos_ab > datos_ac && datos_ab
                    > datos_bc, true);
            }
        }

        return std::make_pair(false, true);
    });
}
```

```
template<std::uint32_t N>
inline bool es_grobner(const std::array<tbb::concurrent_vector<std
    ::pair<sparse_polynomial, std::uint32_t>>, N + 1>&
    polinomios_entrada, const tbb::concurrent_vector<std::uint32_t>&
     indices, const piramide_polinomial<N>& piramide, const
    grevlex_cache<N>& conv)
{
    if (piramide.get_status(0).second) {
        return true;
    }
std::cerr << "revisando si es grobner en " << piramide.degree( )
    << " con " << indices.size( ) << " irreducibles\n";
    if (std::find_if(polinomios_entrada.begin( ) + piramide.degree(
        ) + 1, polinomios_entrada.end( ), base::not_fn(FUNCTOR(base
        ::empty))) != polinomios_entrada.end( )) {
        return false;
    }

    tbb::task_group_context contexto;
    tbb::parallel_for(std::uint32_t(0), std::uint32_t(indices.size(
        )), [&](auto i) {
        auto indice1 = indices[i];
        auto lider1 = conv.monomial_of(indice1);
        auto clave1 = piramide.get_status(indice1).first;

        if (lider1.degree( ) == piramide.degree( )) {
            return (void)contexto.cancel_group_execution( );
        }

        tbb::parallel_for(std::uint32_t(0), i, [&](auto j) {
            auto indice2 = indices[j];
            auto lider2 = conv.monomial_of(indice2);
            auto clave2 = piramide.get_status(indice2).first;
```

```
            auto mcm = least_common_multiple(lider1, lider2);
            auto grado_mcm = mcm.degree();

            if (!is_coprime(lider1, lider2) && grado_mcm > piramide.
               degree() && !criterio_buchberger(mcm, lider1, clave1,
               lider2, clave2, piramide, conv)) {
               return (void)contexto.cancel_group_execution();
            }
         }, tbb::auto_partitioner(), contexto);
      }, tbb::auto_partitioner(), contexto);

      return !contexto.is_group_execution_cancelled();
}

template<std::uint32_t N>
inline std::pair<tbb::concurrent_vector<std::pair<
      sparse_polynomial, std::uint32_t>>, bool> grobner_interrumpible(
      const std::array<tbb::concurrent_vector<std::pair<
      sparse_polynomial, std::uint32_t>>, N + 1>& polinomios_entrada,
      std::uint32_t lineales_esperados, grevlex_cache<N>& conv)
{
      std::uint32_t lineales = 0;
      piramide_polinomial<N> piramide;
      tbb::concurrent_vector<std::uint32_t> indices;
      tbb::concurrent_vector<std::uint32_t> indices_eliminar;

      for (std::uint32_t d = 0; d <= N; ++d) {
         conv.emplace_degree(d);
         piramide.increase_degree(conv);
std::cerr << "***␣piso␣" << d << "...\n";
         std::uint32_t cola = 0;
         std::uint32_t existentes = indices.size();
         tbb::concurrent_vector<std::pair<std::uint32_t, std::
            uint32_t>> inducidos_pendientes;
         tbb::concurrent_vector<std::pair<std::uint32_t, std::
```

```
        uint32_t>> multiplos_pendientes;
std::cerr << "importando entrada de " << polinomios_entrada[d].
   size() << " polinomios\n";
      actualiza_base(polinomios_entrada[d], inducidos_pendientes,
         multiplos_pendientes, indices, indices_eliminar, lineales,
          piramide, conv);

      while (cola < indices.size() && lineales <
         lineales_esperados) {
         tbb::parallel_for(cola, std::uint32_t(indices.size()),
            [&](auto i) {
            auto indice1 = indices[i];
            auto lider1 = conv.monomial_of(indice1);
            auto clave1 = piramide.get_status(indice1).first;

            auto grado = lider1.degree();
            auto grado_inducido = std::min(grado + 1, N);

            if (grado_inducido <= d && (grado_inducido == d || i
               >= existentes)) {
               tbb::parallel_for(std::uint32_t(0), grado, [&](auto
                   i) {
                  inducidos_pendientes.emplace_back(indice1, conv.
                     index_of(genera_factor<N>(lider1[i])));
               });
            }

            tbb::parallel_for(std::uint32_t(0), i, [&](auto j) {
               auto indice2 = indices[j];
               auto lider2 = conv.monomial_of(indice2);
               auto clave2 = piramide.get_status(indice2).first;

               auto mcm = least_common_multiple(lider1, lider2);
               auto grado_mcm = mcm.degree();
```

```
                if (!is_coprime(lider1, lider2) && grado_mcm <= d
                    && (grado_mcm == d || i >= existentes || j >=
                    existentes) && !criterio_buchberger(mcm, lider1,
                    clave1, lider2, clave2, piramide, conv)) {
                    multiplos_pendientes.emplace_back(indice1, conv.
                        index_of(mcm / lider1));
                    multiplos_pendientes.emplace_back(indice2, conv.
                        index_of(mcm / lider2));
                }
            });
        });

        simplifica_indices(indices, indices_eliminar, existentes,
            piramide);
        cola = indices.size();
        actualiza_base(tbb::concurrent_vector<std::pair<
            sparse_polynomial, std::uint32_t>>(),
            inducidos_pendientes, multiplos_pendientes, indices,
            indices_eliminar, lineales, piramide, conv);
    }

    if (es_grobner(polinomios_entrada, indices, piramide, conv))
        {
        break;
    }

    if (lineales >= lineales_esperados) {
std::cerr << "interrumpiendo grobner por lineales\n";
        return { piramide.extract_up_to(std::min(piramide.degree(
            ), std::uint32_t(1)), conv), false };
    }
  }

  return { (piramide.diagonalize(indices, conv), piramide.
      extract_up_to(piramide.degree(), conv)), true };
```

```
}

int main( )
{
   constexpr std::uint32_t N = 20;
   constexpr std::uint32_t D = 2;
//tbb::task_scheduler_init mt(1);
   grevlex_cache<N> conv(D);
   std::array<tbb::concurrent_vector<std::pair<sparse_polynomial,
      std::uint32_t>>, N + 1> polinomios_entrada;

   lectura: {
      std::uint32_t capacidad = conv.population_up_to(D);
      std::vector<monomial<N>> trabajo;
      polynomial buffer(capacidad);

      while (get_polynomial(std::cin, trabajo)) {
         lee_polinomio(buffer, trabajo.begin( ), trabajo.end( ),
            conv);
         auto indice_lider = buffer.find_leader(capacidad);

         if (indice_lider != std::uint32_t(-1)) {
            polinomios_entrada[conv.degree_of(indice_lider)].
               emplace_back(sparse_polynomial(buffer, indice_lider
               + 1), indice_lider);
            buffer.clear(indice_lider + 1);
         }
      }
   }
auto t0 = std::chrono::high_resolution_clock::now( );
   auto res = grobner_interrumpible(polinomios_entrada, std::ceil(
      std::sqrt(N)), conv);
   if (!res.second) {
      tbb::parallel_for_each(res.first, [&](auto& comprimido) {
         polinomios_entrada[conv.degree_of(comprimido.second)].
```

```
                push_back(std::move(comprimido));
        });

        res = grobner_interrumpible(polinomios_entrada, N + 2, conv)
            ;
    }
auto t1 = std::chrono::high_resolution_clock::now();
std::cerr << std::chrono::duration_cast<std::chrono::milliseconds
    >(t1 - t0).count() / 1000.0 << '\n';
    escritura: {
        tbb::parallel_sort(res.first.begin(), res.first.end(),
            [&](const auto& comprimido1, const auto& comprimido2) {
            return comprimido1.second < comprimido2.second;
        });

        std::vector<monomial<N>> trabajo;
        std::for_each(res.first.rbegin(), res.first.rend(), [&](
            auto& comprimido) {
            escribe_polinomio(comprimido.first, std::back_inserter(
                trabajo), conv);
            std::reverse(trabajo.begin(), trabajo.end());
            write_polynomial(std::cout, trabajo) << '\n';

            trabajo.clear();
        });
    }
}
```

Bibliografía

[1] Bardet, M., J. C. Faugère, B. Salvy y P. J. Spaenlehauer: *On the Complexity of Solving Quadratic Boolean Systems*. J. Complex., 29(1):53–75, Feb. 2013, ISSN 0885-064X.

[2] Bayer, D. y M. Stillman: *A theorem on refining division orders by the reverse lexicographic order*. Duke Math. J., 55(2):321–328, Jun. 1987.

[3] Bertsimas, D., G. Perakis y S. Tayur: *A New Algebraic Geometry Algorithm for Integer Programming*. Manage. Sci., 46(7):999–1008, Jul. 2000, ISSN 0025-1909.

[4] Bosma, W., J. Cannon y C. Playoust: *The Magma algebra system. I. The user language*. J. Symbolic Comput., 24(3-4):235–265, 1997, ISSN 0747-7171.

[5] Bosma, W., J. Cannon y C. Playoust: *Magma Calculator*. 2016. `http://magma.maths.usyd.edu.au/calc/`, visitado el 2016-30-07.

[6] Bosma, W., J. Cannon y C. Playoust: *Magma Computer Algebra Documentation*, 2016. `https://magma.maths.usyd.edu.au/magma/handbook/text/1207`, visitado el 2016-09-28.

[7] Brickenstein, M. y A. Dreyer: *PolyBoRi: A framework for Gröbner Basis Computations with Boolean Polynomials*. Journal of Symbolic Computation, 44(9):1326 – 1345, 2009, ISSN 0747-7171. Effective Methods in Algebraic Geometry.

[8] Buchberger, B.: *A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases*. En *Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, EUROSAM '79, págs. 3–21, London, UK, UK, 1979. Springer-Verlag, ISBN 3-540-09519-5.

[9] Buchberger, B.: *An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal*. Tesis de Doctorado, 2006.

[10] Campos, R. A. C., F. D. S. Troncoso y F. J. Z. Martínez: *A cache-aware data structure for representing boolean polynomials*. En *12th International Conference on Electrical Engineering, Computing Science and Automatic Control, CCE 2015, Mexico City, Mexico, October 28-30, 2015*, págs. 1–5, 2015.

[11] Campos, R. A. C., F. D. S. Troncoso y F. J. Z. Martínez: *High Performance Computing*. En *Communications in Computer and Information Science*, vol. 697. Springer, 2017.

[12] Coladon, T.: *OpenF4: Library for Gröebner basis computations over finite fields*. `http://nauotit.github.io/openf4/`, 2016.

[13] Cook, S. A.: *The Complexity of Theorem-proving Procedures*. En *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, págs. 151–158, New York, NY, USA, 1971. ACM.

[14] Courtois, N., A. Klimov, J. Patarin y A. Shamir: *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations*. En *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, págs. 392–407, Berlin, Heidelberg, 2000. Springer-Verlag, ISBN 3-540-67517-5.

[15] Cox, D. A., J. Little y D. O'Shea: *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007, ISBN 0387356509.

[16] Developers, T. S.: *SageMath, the Sage Mathematics Software System*, 2016. `http://www.sagemath.org`.

[17] Eder, C.: *Signature–based algorithms to compute standard bases*. Tesis de Doctorado, Technische Universität Kaiserslautern, 2012.

[18] Eder, C.: *An Analysis of Inhomogeneous Signature-based Gröbner Basis Computations*. J. Symb. Comput., 59:21–35, Dic. 2013, ISSN 0747-7171.

[19] Eder, C.: *Predicting Zero Reductions in Gröbner Basis Computations*. En *Proceedings of the 2014 Symposium on Symbolic-Numeric Computation*, SNC '14, págs. 109–110, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2963-7.

[20] Eder, C. y J. C. Faugère: *A Survey on Signature-based GrÖBner Basis Computations*. ACM Commun. Comput. Algebra, 49(2):61–61, Ago. 2015, ISSN 1932-2240.

[21] Faugère, J., P. Gianni, D. Lazard y T. Mora: *Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering*. J. Symb. Comput., 16(4):329–344, Oct. 1993, ISSN 0747-7171.

[22] Faugère, J. C.: *A new efficient algorithm for computing Gröbner bases (F4)*. Journal of Pure and Applied Algebra, 139(1–3):61 – 88, 1999, ISSN 0022-4049.

[23] Faugère, J. C.: *A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F5)*. En *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, págs. 75–83, New York, NY, USA, 2002. ACM, ISBN 1-58113-484-3.

[24] Faugère, J. C.: *FGb: A Library for Computing Gröbner Bases*. En Fukuda, K., J. Hoeven, M. Joswig y N. Takayama (eds.): *Mathematical Software - ICMS 2010*, vol. 6327 de *Lecture Notes in Computer Science*, págs. 84–87, Berlin, Heidelberg, September 2010. Springer Berlin / Heidelberg.

[25] Faugère, J. C., P. Gaudry, L. Huot y G. Renault: *Sub-cubic Change of Ordering for GrÖbner Basis: A Probabilistic Approach*. En *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, págs. 170–177, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2501-1.

[26] Faugère, J. C. y A. Joux: *Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases*, págs. 44–60. Springer Berlin Heidelberg, 2003, ISBN 978-3-540-45146-4.

[27] Fayssal, M.: *Faugère-Lachartre Parallel Gaussian Elimination for Gröbner Bases Computations Over Finite Fields*. Tesis de Licenciatura, Pierre and Marie Curie University, 2012.

[28] Gebauer, R. y H. M. Möller: *On an Installation of Buchberger's Algorithm*. J. Symb. Comput., 6(2-3):275–286, Dic. 1988, ISSN 0747-7171.

[29] Grayson, D. R. y M. E. Stillman: *Macaulay2, a software system for research in algebraic geometry*. `http://www.math.uiuc.edu/Macaulay2/`.

[30] Gurobi Optimization, I.: *Gurobi Optimizer Reference Manual*, 2017. `http://www.gurobi.com`.

[31] Hammarlund, P., A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza y T. Burton: *Haswell: The Fourth-Generation Intel Core Processor*. IEEE Micro, 34(2):6–20, 2014, ISSN 0272-1732.

[32] Hennessy, J. L. y D. A. Patterson: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th ed., 2011, ISBN 012383872X, 9780123838728.

[33] Herrera García, J. L. J.: *Autenticación y Cifrado Basado en Ecuaciones Cuadráticas de Varias Variables*. Tesis de Doctorado, Instituto Politécnico Nacional, 2015.

[34] Hinkelmann, F. y E. Arnold: *Fast Gröbner Basis Computation for Boolean Polynomials*. CoRR, 2010.

[35] Karp, R. M.: *Reducibility among Combinatorial Problems*, págs. 85–103. Springer US, Boston, MA, 1972, ISBN 978-1-4684-2001-2.

[36] Knuth, D. E.: *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th ed., 2009, ISBN 0321580508, 9780321580504.

[37] Lay, D. C.: *Linear Algebra and its applications/*. Addison-Wesley, Boston, 4ª ed., 2012.

[38] Lazard, D.: *Solving Zero-dimensional Algebraic Systems*. J. Symb. Comput., 13(2):117–131, Feb. 1992, ISSN 0747-7171.

[39] Minato, S. i.: *Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems*. En *Proceedings of the 30th International Design Automation Conference*, DAC '93, págs. 272–277, New York, NY, USA, 1993. ACM, ISBN 0-89791-577-1.

[40] Mohamed, M. S. E., D. Cabarcas, J. Ding, J. Buchmann y S. Bulygin: *MXL3: An Efficient Algorithm for Computing Gröbner Bases of Zero-dimensional Ideals*. En *Proceedings of the 12th International Conference on Information Security and Cryptology*, ICISC'09, págs. 87–100, Berlin, Heidelberg, 2010. Springer-Verlag, ISBN 3-642-14422-5, 978-3-642-14422-6.

[41] Neumann, S.: *A modified parallel F4 algorithm for shared and distributed memory architectures.* En Kovacs, L. y T. Kutsia (eds.): *SCSS 2013. 5th International Symposium on Symbolic Computation in Software Science*, vol. 15 de *EPiC Series in Computing*, págs. 70–80. EasyChair, 2013.

[42] Patarin, J.: *Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms.* Springer Berlin Heidelberg, 1996, ISBN 978-3-540-68339-1.

[43] Roune, B. H. y M. Stillman: *Practical Gröbner Basis Computation.* En *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, ISSAC '12, págs. 203–210, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1269-1.

[44] Steel, A.: *A Dense Variant of the F4 Gröbner Basis Algorithm*, 2013. `http://magma.maths.usyd.edu.au/~allan/densef4/`, visitado el 2016-30-07.

[45] Steel, A.: *Direct Solution of the (11,9,8)-MinRank Problem by the Block Wiedemann Algorithm in Magma with a Tesla GPU.* En *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*, PASCO '15, págs. 2–6, New York, NY, USA, 2015. ACM, ISBN 978-1-4503-3599-7. `http://doi.acm.org/10.1145/2790282.2791392`.

[46] Warners, J. P.: *A Linear-time Transformation of Linear Inequalities into Conjunctive Normal Form.* Inf. Process. Lett., 68(2):63–69, Oct. 1998, ISSN 0020-0190.

[47] Woeginger, G. J.: *Combinatorial Optimization - Eureka, You Shrink!* cap. Exact Algorithms for NP-hard Problems: A Survey, págs. 185–207. Springer-Verlag New York, Inc., New York, NY, USA, 2003, ISBN 3-540-00580-3.